

0.1 Converting between Types

Sometimes all the operations "stay within a type".

- 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
 - * Class to built-in type (**int**, **char**, etc.)
 - Must be non-**static** member function; Cannot be **friend**
 - Do not specify return type; implicitly returns type to which you are converting
- Example
 - Prototype
 - * **A::operator char *() const;**
 - * Casts class **A** to a temporary **char ***
 - * **(char *)s** calls **s.operator char*()**
 - Also, overloaded cast-operator functions can be defined for converting objects of user-defined types into built-in types or into objects of other user-defined types.
 - * **A::operator int() const;**
 - * **A::operator OtherClass() const;**
- 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 ***
 - * Do not have to overload **<<**

0.2 Case Study: A String Class

- Build class String
 - To handle String creation, manipulation
 - Class **string** in standard library (more Chapter 15)
- Conversion constructor
 - Single-argument constructor
 - Turns objects of other types into class objects
 - * **String s1("hi");**
 - * Creates a **String** from a **char ***
 - Any single-argument constructor is a conversion constructor

The programs of Figs. 1-10 demonstrates the building of our own **String** class to handle the creation and manipulation of strings.

```

1 // Fig. 8.7: string1.h
2 // String class definition.
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
13     friend istream &operator>>( istream
14
15 public:
16     String( const char * = "" ); // conversion/default ctor
17     String( const String & ); // copy constructor
18     ~String(); // destructor
19
20     const String &operator=( const String & ); //
21     const String &operator+=( const String & ); //
22
23     bool operator!() const; // i
24     bool operator==( const String & ) const; // t
25     bool operator<( const String & ) const; // t
26

```

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

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

Can also concatenate a **String** and a **char *** because the compiler will cast the **char *** argument to a **String**. However, it can only do 1 level of casting.

© 2003 Prentice Hall, Inc. All rights reserved.

```

27 // test s1 != s2
28 bool operator!=( const String &right ) const
29 {
30     return !( *this == right );
31
32 } // end function operator!=
33
34 // test s1 > s2
35 bool operator>( const String &right ) const
36 {
37     return right < *this;
38
39 } // end function operator>
40
41 // test s1 <= s2
42 bool operator<=( const String &right ) const
43 {
44     return !( right < *this );
45
46 } // end function operator <=
47
48 // test s1 >= s2
49 bool operator>=( const String &right ) const
50 {
51     return !( *this < right );
52
53 } // end function operator>=

```

© 2003 Prentice Hall, Inc. All rights reserved.

Figure 1: **String** class definition with operator overloading. (part 1 of 2)

```

54 char &operator[] ( int ); // s
55 const char &operator[] ( int ) const; // s
57
58 String operator() ( int, int ); // return a substring
59
60 int getLength() const;
61
62 private:
63 int length; // string length
64 char *sPtr; // pointer to start of string
65
66 void setString( const char * ); // utility function
67
68 }; // end class String
69
70 #endif

```

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

Overload the function call operator () to return a substring. This operator can have any amount of operands.

Outline 41
string1.h (3 of 3)

© 2003 Prentice Hall, Inc.
All rights reserved.

Figure 2: **String** class definition with operator overloading. (part 2 of 2)

```

1 // Fig. 8.8: string1.cpp
2 // Member function definitions for class String.
3 #include <iostream>
4
5 using std::cout;
6 using std::endl;
7
8 #include <iomanip>
9
10 using std::setw;
11
12 #include <new> // C++ standard "new" operator
13
14 #include <cstring> // strcpy and strcat prototypes
15 #include <cstdlib> // exit prototype
16
17 #include "string1.h" // String class definition
18
19 // conversion constructor converts char * to String
20 String::String( const char *s )
21     : length( strlen( s ) )
22 {
23     cout << "Conversion constructor: " << s << '\n';
24     setString( s ); // call utility function
25 }
26 // end String conversion constructor

```



[Outline](#)

42

string1.cpp (1 of 8)

© 2003 Prentice Hall, Inc.
All rights reserved.

```

27
28 // copy constructor
29 String::String( const String &copy )
30     : length( copy.length )
31 {
32     cout << "Copy constructor: " << copy.sPtr << '\n';
33     setString( copy.sPtr ); // call utility function
34 }
35 // end String copy constructor
36
37 // destructor
38 String::~String()
39 {
40     cout << "Destructor: " << sPtr << '\n';
41     delete [] sPtr; // reclaim string
42 }
43 // end ~String destructor
44
45 // overloaded = operator; avoids self assignment
46 const String &String::operator=( const String &right )
47 {
48     cout << "operator= called\n";
49
50     if ( &right != this ) { // avoid self assignment
51         delete [] sPtr; // prevents memory leak
52         length = right.length; // new String length
53         setString( right.sPtr ); // call utility function
54     }

```



[Outline](#)

43

string1.cpp (2 of 8)

© 2003 Prentice Hall, Inc.
All rights reserved.

Figure 3: **String** class member-function and **friend**-function definition. (part 1 of 4)

```

55
56     else
57         cout << "Attempted assignment of a String to itself\n";
58
59     return *this; // enables cascaded assignments
60
61 } // end function operator=
62
63 // concatenate right operand to this object and
64 // store in this object.
65 const String &String::operator+=( const String &right )
66 {
67     size_t newLength = length + right.length; // new length
68     char *tempPtr = new char[ newLength + 1 ]; // create memory
69
70     strcpy( tempPtr, sPtr ); // copy sPtr
71     strcpy( tempPtr + length, right.sPtr ); // copy right.sPtr
72
73     delete [] sPtr; // reclaim old space
74     sPtr = tempPtr; // assign new array to sPtr
75     length = newLength; // assign new length to length
76
77     return *this; // enables cascaded calls
78
79 } // end function operator+=
80

```



[Outline](#)

44

string1.cpp (3 of 8)

© 2003 Prentice Hall, Inc.
All rights reserved.

```

81 // is this String empty?
82 bool String::operator!() const
83 {
84     return length == 0;
85
86 } // end function operator!
87
88 // is this String equal to right String?
89 bool String::operator==( const String &right ) const
90 {
91     return strcmp( sPtr, right.sPtr ) == 0;
92
93 } // end function operator==
94
95 // is this String less than right String?
96 bool String::operator<( const String &right ) const
97 {
98     return strcmp( sPtr, right.sPtr ) < 0;
99
100 } // end function operator<
101

```



[Outline](#)

45

string1.cpp (4 of 8)

© 2003 Prentice Hall, Inc.
All rights reserved.

Figure 4: **String** class member-function and **friend**-function definition. (part 2 of 4)

```

102 // return reference to character in String as lvalue
103 char &String::operator[]( int subscript )
104 {
105     // test for subscript out of range
106     if ( subscript < 0 || subscript >= length ) {
107         cout << "Error: Subscript " << subscript
108             << " out of range" << endl;
109
110         exit( 1 ); // terminate program
111     }
112
113     return sPtr[ subscript ]; // creates lvalue
114 }
115 // end function operator[]
116
117 // return reference to character in String as rvalue
118 const char &String::operator[]( int subscript ) const
119 {
120     // test for subscript out of range
121     if ( subscript < 0 || subscript >= length ) {
122         cout << "Error: Subscript " << subscript
123             << " out of range" << endl;
124
125         exit( 1 ); // terminate program
126     }
127
128     return sPtr[ subscript ]; // creates rvalue
129 }
130 // end function operator[]

```



[Outline](#)

46

string1.cpp (5 of 8)

© 2003 Prentice Hall, Inc.
All rights reserved.

```

131
132 // return a substring beginning at index and
133 // of length subLength
134 String String::operator()( int index, int subLength )
135 {
136     // if index is out of range or substring length < 0,
137     // return an empty String object
138     if ( index < 0 || index >= length || subLength < 0 )
139         return ""; // converted to a String object automatically
140
141     // determine length of substring
142     int len;
143
144     if ( ( subLength == 0 ) || ( index + subLength > length ) )
145         len = length - index;
146     else
147         len = subLength;
148
149     // allocate temporary array for substring and
150     // terminating null character
151     char *tempPtr = new char[ len + 1 ];
152
153     // copy substring into char array and terminate string
154     strncpy( tempPtr, &sPtr[ index ], len );
155     tempPtr[ len ] = '\0';

```



[Outline](#)

47

string1.cpp (6 of 8)

© 2003 Prentice Hall, Inc.
All rights reserved.

Figure 5: **String** class member-function and **friend**-function definition.
(part 3 of 4)


```

156
157 // create temporary String object containing the substring
158 String tempString( tempPtr );
159 delete [] tempPtr; // delete temporary array
160
161 return tempString; // return copy of the temporary String
162
163 } // end function operator()
164
165 // return string length
166 int String::getLength() const
167 {
168     return length;
169 }
170 // end function getLenth
171
172 // utility function called by constructors and operator=
173 void String::setString( const char *string2 )
174 {
175     sPtr = new char[ length + 1 ]; // allocate memory
176     strcpy( sPtr, string2 ); // copy literal to object
177 }
178 // end function setString

```



[Outline](#)

48



string1.cpp (7 of 8)

© 2003 Prentice Hall, Inc.
All rights reserved.

```

179
180 // overloaded output operator
181 ostream &operator<<( ostream &output, const String &s )
182 {
183     output << s.sPtr;
184
185     return output; // enables cascading
186 }
187 // end function operator<<
188
189 // overloaded input operator
190 istream &operator>>( istream &input, String &s )
191 {
192     char temp[ 100 ]; // buffer to store input
193
194     input >> setw( 100 ) >> temp;
195     s = temp; // use String class assignment operator
196
197     return input; // enables cascading
198 }
199 // end function operator>>

```



[Outline](#)

49



string1.cpp (8 of 8)

© 2003 Prentice Hall, Inc.
All rights reserved.

Figure 6: **String** class member-function and **friend**-function definition.
(part 4 of 4)


```

1 // Fig. 8.9: fig08_09.cpp
2 // String class test program.
3 #include <iostream>
4
5 using std::cout;
6 using std::endl;
7
8 #include "string1.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 << "\"\n";
19     << "\n\nThe results of comparing s2 and s1:"
20     << "\ns2 == s1 yields "
21     << ( s2 == s1 ? "true" : "false" )
22     << "\ns2 != s1 yields "
23     << ( s2 != s1 ? "true" : "false" )
24     << "\ns2 > s1 yields "
25     << ( s2 > s1 ? "true" : "false" )

```



[Outline](#)

50

fig08_09.cpp
(1 of 4)

© 2003 Prentice Hall, Inc.
All rights reserved.

```

26     << "\ns2 < s1 yields "
27     << ( s2 < s1 ? "true" : "false" )
28     << "\ns2 >= s1 yields "
29     << ( s2 >= s1 ? "true" : "false" )
30     << "\ns2 <= s1 yields "
31     << ( s2 <= s1 ? "true" : "false" );
32
33     // test overloaded String empty (!) operator
34     cout << "\n\nTesting !s3:\n";
35
36     if ( !s3 ) {
37         cout << "s3 is empty; assigning s1 to s3;\n";
38         s3 = s1; // test overloaded assignment
39         cout << "s3 is \"" << s3 << "\"\n";
40     }
41
42     // test overloaded String concatenation operator
43     cout << "\n\ns1 += s2 yields s1 = ";
44     s1 += s2; // test overloaded concatenation
45     cout << s1;
46
47     // test conversion constructor
48     cout << "\n\ns1 += \" to you\" yields\n";
49     s1 += " to you"; // test conversion constructor
50     cout << "s1 = " << s1 << "\n\n";

```



[Outline](#)

51

fig08_09.cpp
(2 of 4)

© 2003 Prentice Hall, Inc.
All rights reserved.

Figure 7: **String** class test program. (part 1 of 2)

```

51
52 // test overloaded function call operator () for substring
53 cout << "The substring of s1 starting at\n"
54     << "location 0 for 14 characters, s1(0, 14), is:\n"
55     << s1( 0, 14 ) << "\n\n";
56
57 // test substring "to-end-of-String" option
58 cout << "The substring of s1 starting at\n"
59     << "location 15, s1(15, 0), is: "
60     << s1( 15, 0 ) << "\n\n"; // 0 is "to end of string"
61
62 // test copy constructor
63 String *s4Ptr = new String( s1 );
64 cout << "\n*s4Ptr = " << *s4Ptr << "\n\n";
65
66 // test assignment (=) operator with self-assignment
67 cout << "assigning *s4Ptr to *s4Ptr\n";
68 *s4Ptr = *s4Ptr; // test overloaded assignment
69 cout << "*s4Ptr = " << *s4Ptr << '\n';
70
71 // test destructor
72 delete s4Ptr;
73

```



[Outline](#)

52

fig08_09.cpp
(3 of 4)

© 2003 Prentice Hall, Inc.
All rights reserved.

```

74 // test using subscript operator to create lvalue
75 s1[ 0 ] = 'H';
76 s1[ 6 ] = 'B';
77 cout << "\ns1 after s1[0] = 'H' and s1[6] = 'B' is: "
78     << s1 << "\n\n";
79
80 // test subscript out of range
81 cout << "Attempt to assign 'd' to s1[30] yields:" << endl;
82 s1[ 30 ] = 'd'; // ERROR: subscript out of range
83
84 return 0;
85
86 } // end main

```



[Outline](#)

53

fig08_09.cpp
(4 of 4)

© 2003 Prentice Hall, Inc.
All rights reserved.

Figure 8: **String** class test program. (part 2 of 2)

```

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

```

The constructor and destructor are called for the temporary `String` (converted from the `char *` "to you").



Outline

54

fig08_09.cpp
(1 of 3)

© 2003 Prentice Hall, Inc.
All rights reserved.

```

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

```



Outline

55

fig08_09.cpp
(2 of 3)

© 2003 Prentice Hall, Inc.
All rights reserved.

Figure 9: `String` class test program, output. (part 1 of 2)

```
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](#)

fig08_09.cpp
(3 of 3)

56

© 2003 Prentice Hall, Inc.
All rights reserved.

Figure 10: **String** class test program, output. (part 2 of 2)

0.3 Overloading ++ and –

- Increment/decrement operators can be overloaded
 - Add 1 to a **Date** object, **d1**
 - Prototype (member function)
 - * **Date &operator++()**;
 - * **++d1** same as **d1.operator++()**
 - Prototype (non-member)
 - * **Friend Date &operator++(Date &)**;
 - * **++d1** same as **operator++(d1)**
- To distinguish pre/post increment
 - Post increment has a dummy parameter; **int** of **0**
 - Prototype (member function)
 - * **Date operator++(int);**

- * `d1++` same as `d1.operator++(0)`
 - Prototype (non-member)
 - * `friend Date operator++(Date &, int);`
 - * `d1++` same as `operator++(d1, 0)`
 - Integer parameter does not have a name; not even in function definition
- Return values
 - Preincrement
 - * Returns by reference (`Date &`)
 - * lvalue (can be assigned)
 - Postincrement
 - * Returns by value
 - * Returns temporary object with old value
 - * rvalue (cannot be on left side of assignment)
- Decrement operator analogous

0.4 Case Study: A Date Class

- Example Date class. The class uses overloaded preincrement and postincrement operators to add 1 to the day in a `Date` object, while causing appropriate increments to the month and year if necessary.
 - 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

```

1 // Fig. 8.10: date1.h
2 // Date class definition.
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 = 1 );
14     void setDate( int, int, int ); // set date
15
16     Date &operator++();           // preincrement operator
17     Date operator++( int );      // postincrement operator
18
19     const Date &operator+=( int ); // add days, modify object
20
21     bool leapYear( int ) const;   // is this a leap year?
22     bool endOfMonth( int ) const; // is this end of month?

```

Note difference between pre and post increment.



Outline

61

date1.h (1 of 2)

© 2003 Prentice Hall, Inc.
All rights reserved.

```

23
24 private:
25     int month;
26     int day;
27     int year;
28
29     static const int days[]; // array of days per month
30     void helpIncrement();    // utility function
31
32 }; // end class Date
33
34 #endif

```



Outline

62

date1.h (2 of 2)

© 2003 Prentice Hall, Inc.
All rights reserved.

Figure 11: **Date** class definition with overloaded increment operator.

```

1 // Fig. 8.11: date1.cpp
2 // Date class member function definitions.
3 #include <iostream>
4 #include "date1.h"
5
6 // initialize static member at file scope;
7 // one class-wide copy
8 const int Date::days[] =
9     { 0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 };
10
11 // Date constructor
12 Date::Date( int m, int d, int y )
13 {
14     setDate( m, d, y );
15 }
16 // end Date constructor
17
18 // set month, day and year
19 void Date::setDate( int mm, int dd, int yy )
20 {
21     month = ( mm >= 1 && mm <= 12 ) ? mm : 1;
22     year = ( yy >= 1900 && yy <= 2100 ) ? yy : 1900;
23 }

```



Outline

63

date1.cpp (1 of 5)

© 2003 Prentice Hall, Inc.
All rights reserved.

```

24 // test for a leap year
25 if ( month == 2 && leapYear( year ) )
26     day = ( dd >= 1 && dd <= 29 ) ? dd : 1;
27 else
28     day = ( dd >= 1 && dd <= days[ month ] ) ? dd : 1;
29
30 // end function setDate
31
32 // overloaded preincrement operator
33 Date &Date::operator++()
34 {
35     helpIncrement();
36
37     return *this; // reference return to create an lvalue
38 }
39 // end function operator++
40 // overloaded postincrement operator; no
41 // integer parameter does not have a par
42 Date Date::operator++( int )
43 {
44     Date temp = *this; // hold current s
45     helpIncrement();
46
47     // return unincremented, saved, tempo
48     return temp; // value return; not a
49 }
50
51 // end function operator++

```

Postincrement 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.

Also note that the integer parameter does not have a name.



Outline

64

date1.cpp (2 of 5)

© 2003 Prentice Hall, Inc.
All rights reserved.

Figure 12: **Date** class member- and **friend**-function definition. (part 1 of 3)


```

52
53 // add specified number of days to date
54 const Date &Date::operator+=( int additionalDays )
55 {
56     for ( int i = 0; i < additionalDays; i++ )
57         helpIncrement();
58
59     return *this;    // enables cascading
60
61 } // end function operator+=
62
63 // if the year is a leap year, return true;
64 // otherwise, return false
65 bool Date::leapYear( int testYear ) const
66 {
67     if ( testYear % 400 == 0 ||
68         ( testYear % 100 != 0 && testYear % 4 == 0 ) )
69         return true;    // a leap year
70     else
71         return false; // not a leap year
72
73 } // end function leapYear
74

```



[Outline](#)

65

date1.cpp (3 of 5)

© 2003 Prentice Hall, Inc.
All rights reserved.

```

75 // determine whether the day is the last day of the month
76 bool Date::endOfMonth( int testDay ) const
77 {
78     if ( month == 2 && leapYear( year ) )
79         return testDay == 29; // last day of Feb. in leap year
80     else
81         return testDay == days[ month ];
82
83 } // end function endOfMonth
84
85 // function to help increment the date
86 void Date::helpIncrement()
87 {
88     // day is not end of month
89     if ( !endOfMonth( day ) )
90         ++day;
91
92     else
93
94         // day is end of month and month < 12
95         if ( month < 12 ) {
96             ++month;
97             day = 1;
98         }
99

```



[Outline](#)

66

date1.cpp (4 of 5)

© 2003 Prentice Hall, Inc.
All rights reserved.

Figure 13: **Date** class member-and **friend**-function definition. (part 2 of 3)

```

100     // last day of year
101     else {
102         ++year;
103         month = 1;
104         day = 1;
105     }
106
107 } // end function helpIncrement
108
109 // overloaded output operator
110 ostream &operator<<( ostream &output, const Date &d )
111 {
112     static char *monthName[ 13 ] = { "", "January",
113         "February", "March", "April", "May", "June",
114         "July", "August", "September", "October",
115         "November", "December" };
116
117     output << monthName[ d.month ] << ' '
118         << d.day << ", " << d.year;
119
120     return output; // enables cascading
121
122 } // end function operator<<

```



[Outline](#)

67

date1.cpp (5 of 5)

© 2003 Prentice Hall, Inc.
All rights reserved.

Figure 14: **Date** class member-and **friend**-function definition. (part 3 of 3)

```

1 // Fig. 8.12: fig08_12.cpp
2 // Date class test program.
3 #include <iostream>
4
5 using std::cout;
6 using std::endl;
7
8 #include "date1.h" // Date class definition
9
10 int main()
11 {
12     Date d1; // defaults to January 1, 1900
13     Date d2( 12, 27, 1992 );
14     Date d3( 0, 99, 8045 ); // invalid date
15
16     cout << "d1 is " << d1 << "\nd2 is " << d2
17         << "\nd3 is " << d3;
18
19     cout << "\nd2 += 7 is " << ( d2 += 7 );
20
21     d3.setDate( 2, 28, 1992 );
22     cout << "\n\n d3 is " << d3;
23     cout << "\n++d3 is " << ++d3;
24
25     Date d4( 7, 13, 2002 );

```



[Outline](#)

68

fig08_12.cpp
(1 of 2)

© 2003 Prentice Hall, Inc.
All rights reserved.

```

26
27     cout << "\n\nTesting the preincrement operator:\n"
28         << " d4 is " << d4 << '\n';
29     cout << "++d4 is " << ++d4 << '\n';
30     cout << " d4 is " << d4;
31
32     cout << "\n\nTesting the postincrement operator:\n"
33         << " d4 is " << d4 << '\n';
34     cout << "d4++ is " << d4++ << '\n';
35     cout << " d4 is " << d4 << endl;
36
37     return 0;
38
39 } // end main

```



[Outline](#)

69

fig08_12.cpp
(2 of 2)

© 2003 Prentice Hall, Inc.
All rights reserved.

Figure 15: **Date** class test program.

```
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 July 13, 2002
++d4 is July 14, 2002
d4 is July 14, 2002

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



[Outline](#)

70

fig08_12.cpp
output (1 of 1)

© 2003 Prentice Hall, Inc.
All rights reserved.

Figure 16: **Date** class test program, output.

0.5 Standard Library Classes **string** and **vector**

We learned that we can build a **String** (**Array**) class that is better than the C-style, **char *** strings (pointer-based arrays) that C++ absorbed from C.

- Classes built into C++
 - Available for anyone to use
 - **string** ; Similar to our **String** class
 - **vector**; Dynamically resizable array
- Redo our **String** and **Array** examples
 - Use **string** and **vector**
- Class **string**
 - Header **<string>**, namespace **std**
 - Can initialize **string s1("hi");**

- Overloaded <<; **cout** << **s1**
- Overloaded relational operators; **==** **!=** **>=** **>** **<=** **<**
- Assignment operator **=**
- Concatenation (overloaded **+=**)
- Substring function **substr**
 - * **s1.substr(0, 14)**; ; Starts at location 0, gets 14 characters
 - * **S1.substr(15)** ; Substring beginning at location 15
- Overloaded **[]**
 - * Access one character
 - * No range checking (if subscript invalid)
- **at** function
 - * **s1.at(10)**
 - * Character at subscript 10
 - * Has bounds checking; will end program if invalid (learn more in Chapter 13)

The programs of Figs. 17-19 reimplements the program of Figs. 7-10, using standart class **string**.

```

1 // Fig. 8.13: fig08_13.cpp
2 // Standard library string class test program.
3 #include <iostream>
4
5 using std::cout;
6 using std::endl;
7
8 #include <string>
9
10 using std::string;
11
12 int main()
13 {
14     string s1( "happy" );
15     string s2( " birthday" );
16     string s3;
17
18     // test overloaded equality and relational operators
19     cout << "s1 is \" << s1 << "\", s2 is \" << s2
20          << "\", s3 is \" << s3 << "\n";
21     << "\n\nThe results of comparing s2 and s1:"
22     << "\ns2 == s1 yields "
23     << ( s2 == s1 ? "true" : "false" )
24     << "\ns2 != s1 yields "
25     << ( s2 != s1 ? "true" : "false" )

```



[Outline](#)

74

fig08_13.cpp
(1 of 4)

© 2003 Prentice Hall, Inc.
All rights reserved.

```

26     << "\ns2 > s1 yields "
27     << ( s2 > s1 ? "true" : "false" )
28     << "\ns2 < s1 yields "
29     << ( s2 < s1 ? "true" : "false" )
30     << "\ns2 >= s1 yields "
31     << ( s2 >= s1 ? "true" : "false" )
32     << "\ns2 <= s1 yields "
33     << ( s2 <= s1 ? "true" : "false" );
34
35     // test string member function empty
36     cout << "\n\nTesting s3.empty():\n";
37
38     if ( s3.empty() ) {
39         cout << "s3 is empty; assigning s1 to s3;\n";
40         s3 = s1; // assign s1 to s3
41         cout << "s3 is \" << s3 << "\n";
42     }
43
44     // test overloaded string concatenation operator
45     cout << "\n\ns1 += s2 yields s1 = ";
46     s1 += s2; // test overloaded concatenation
47     cout << s1;
48

```



[Outline](#)

75

fig08_13.cpp
(2 of 4)

© 2003 Prentice Hall, Inc.
All rights reserved.

Figure 17: Standart library class **string** (part 1 of 2).

```

49 // test overloaded string concatenation operator
50 // with C-style string
51 cout << "\n\ns1 += \" to you\" yields\n";
52 s1 += " to you";
53 cout << "s1 = " << s1 << "\n\n";
54
55 // test string member function substr
56 cout << "The substring of s1 starting at location 0 for\n"
57 << "14 characters, s1.substr(0, 14), is:\n"
58 << s1.substr( 0, 14 ) << "\n\n";
59
60 // test substr "to-end-of-string" option
61 cout << "The substring of s1 starting at\n"
62 << "location 15, s1.substr(15), is:\n"
63 << s1.substr( 15 ) << '\n';
64
65 // test copy constructor
66 string *s4Ptr = new string( s1 );
67 cout << "\n*s4Ptr = " << *s4Ptr << "\n\n";
68
69 // test assignment (=) operator with self-assignment
70 cout << "assigning *s4Ptr to *s4Ptr\n";
71 *s4Ptr = *s4Ptr;
72 cout << "*s4Ptr = " << *s4Ptr << '\n';
73

```



[Outline](#)

76

fig08_13.cpp
(3 of 4)

© 2003 Prentice Hall, Inc.
All rights reserved.

```

74 // test destructor
75 delete s4Ptr;
76
77 // test using subscript operator to create lvalue
78 s1[ 0 ] = 'H';
79 s1[ 6 ] = 'B';
80 cout << "\ns1 after s1[0] = 'H' and s1[6] = 'B' is: "
81 << s1 << "\n\n";
82
83 // test subscript out of range with string member function "at"
84 cout << "Attempt to assign 'd' to s1.at( 30 ) yields:" << endl;
85 s1.at( 30 ) = 'd'; // ERROR: subscript out of range
86
87 return 0;
88
89 } // end main

```



[Outline](#)

77

fig08_13.cpp
(4 of 4)

© 2003 Prentice Hall, Inc.
All rights reserved.

Figure 18: Standard library class `string` (part 2 of 2).


```
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
```



[Outline](#)

78

fig08_13.cpp
output (1 of 2)

© 2003 Prentice Hall, Inc.
All rights reserved.

```
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](#)

79

fig08_13.cpp
output (2 of 2)

© 2003 Prentice Hall, Inc.
All rights reserved.

Figure 19: Standard library class **string**, output.

- Class `vector`
 - Header `<vector>`, namespace `std`
 - Store any type; `vector< int > myArray(10)`
 - Function `size (myArray.size())`
 - Overloaded `[]`; get specific element, `myArray[3]`
 - Overloaded `!=`, `==`, and `=`; inequality, equality, assignment

The programs of Figs. 20-23 reimplements the program of Figs. ??-??, using standard class `vector`.

```

1 // Fig. 8.14: fig08_14.cpp
2 // Demonstrating standard library class vector.
3 #include <iostream>
4
5 using std::cout;
6 using std::cin;
7 using std::endl;
8
9 #include <iomanip>
10
11 using std::setw;
12
13 #include <vector>
14
15 using std::vector;
16
17 void outputVector( const vector< int > & );
18 void inputVector( vector< int > & );
19
20 int main()
21 {
22     vector< int > integers1( 7 ); // 7-element vector< int >
23     vector< int > integers2( 10 ); // 10-element vector< int >
24

```



[Outline](#)

81

fig08_14.cpp
(1 of 5)

© 2003 Prentice Hall, Inc.
All rights reserved.

```

25 // print integers1 size and contents
26 cout << "Size of vector integers1 is "
27     << integers1.size()
28     << "\nvector after initialization:\n";
29 outputVector( integers1 );
30
31 // print integers2 size and contents
32 cout << "\nSize of vector integers2 is "
33     << integers2.size()
34     << "\nvector after initialization:\n";
35 outputVector( integers2 );
36
37 // input and print integers1 and integers2
38 cout << "\nInput 17 integers:\n";
39 inputVector( integers1 );
40 inputVector( integers2 );
41
42 cout << "\nAfter input, the vectors contain:\n"
43     << "integers1:\n";
44 outputVector( integers1 );
45 cout << "integers2:\n";
46 outputVector( integers2 );
47
48 // use overloaded inequality (!=) operator
49 cout << "\nEvaluating: integers1 != integers2\n";
50

```



[Outline](#)

82

fig08_14.cpp
(2 of 5)

© 2003 Prentice Hall, Inc.
All rights reserved.

Figure 20: Standart library class **vector**. (part 1 of 3)

```

51  if ( integers1 != integers2 )
52      cout << "integers1 and integers2 are not equal\n";
53
54  // create vector integers3 using integers1 as an
55  // initializer; print size and contents
56  vector< int > integers3( integers1 ); // copy constructor
57
58  cout << "\nSize of vector integers3 is "
59        << integers3.size()
60        << "\nvector after initialization:\n";
61  outputVector( integers3 );
62
63
64  // use overloaded assignment (=) operator
65  cout << "\nAssigning integers2 to integers1:\n";
66  integers1 = integers2;
67
68  cout << "integers1:\n";
69  outputVector( integers1 );
70  cout << "integers2:\n";
71  outputVector( integers1 );
72

```



[Outline](#)

83

fig08_14.cpp
(3 of 5)

© 2003 Prentice Hall, Inc.
All rights reserved.

```

73  // use overloaded equality (==) operator
74  cout << "\nEvaluating: integers1 == integers2\n";
75
76  if ( integers1 == integers2 )
77      cout << "integers1 and integers2 are equal\n";
78
79  // use overloaded subscript operator to create rvalue
80  cout << "\nintegers1[5] is " << integers1[ 5 ];
81
82  // use overloaded subscript operator to create lvalue
83  cout << "\n\nAssigning 1000 to integers1[5]\n";
84  integers1[ 5 ] = 1000;
85  cout << "integers1:\n";
86  outputVector( integers1 );
87
88  // attempt to use out of range subscript
89  cout << "\n\nAttempt to assign 1000 to integers1.at( 15 )"
90        << endl;
91  integers1.at( 15 ) = 1000; // ERROR: out of range
92
93  return 0;
94
95  } // end main
96

```



[Outline](#)

84

fig08_14.cpp
(4 of 5)

© 2003 Prentice Hall, Inc.
All rights reserved.

Figure 21: Standard library class `vector`. (part 2 of 3)

```
97 // output vector contents
98 void outputVector( const vector< int > &array )
99 {
100     for ( int i = 0; i < array.size(); i++ ) {
101         cout << setw( 12 ) << array[ i ];
102
103         if ( ( i + 1 ) % 4 == 0 ) // 4 numbers per row of output
104             cout << endl;
105     } // end for
106
107     if ( i % 4 != 0 )
108         cout << endl;
109
110 } // end function outputVector
111
112 // input vector contents
113 void inputVector( vector< int > &array )
114 {
115     for ( int i = 0; i < array.size(); i++ )
116         cin >> array[ i ];
117
118 } // end function inputVector
```



[Outline](#)

85

fig08_14.cpp
(5 of 5)

© 2003 Prentice Hall, Inc.
All rights reserved.

Figure 22: Standart library class **vector**. (part 3 of 3)

```

Size of vector integers1 is 7
vector after initialization:
    0    0    0    0
    0    0    0    0

Size of vector integers2 is 10
vector 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 vectors 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](#)

86

fig08_14.cpp
output (1 of 2)

© 2003 Prentice Hall, Inc.
All rights reserved.

```

Size of vector integers3 is 7
vector 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.at( 15 )

abnormal program termination

```



[Outline](#)

87

fig08_14.cpp
output (2 of 2)

© 2003 Prentice Hall, Inc.
All rights reserved.

Figure 23: Standard library class `vector`, output.