IS 0020
Program Design and Software Tools
Introduction to C++ Programming

Operator Overloading
Lecture 6

Feb 10, 2004

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 types (integers, floats, etc.)
- when to use operator overloading?

Fundamentals of Operator Overloading

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

Fundamentals of Operator Overloading

- Using operators on a class object
  - It must be overloaded for that class
  - Exceptions:
    - Assignment operator, **=**
      - May be used without explicit overloading
      - Memberwise assignment between objects
    - Address operator, **&**
      - May be used on any class without overloading
      - Returns address of object
    - Both can be overloaded
- Overloading provides concise notation
  - `object2 = object1.add(object2);`
  - `object2 = object2 + object1;`

Restrictions on Operator Overloading

5

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

Restrictions on Operator Overloading

6

Operators that can be overloaded							
+	-	*	/	%	^	&	|
~	!	=	<	>	+=	-=	*=
/=	%=	^=	&=	|=	<<	>>	>>=
<<=	==	!=	<=	>=	&&	||	++
--	->*	,	->	[]	()	new	delete
new[]	delete[]						

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

Operator Functions As Class Members Vs. As Friend Functions

7

- Operator functions
  - Member functions
    - Use **`this`** keyword to implicitly get argument
    - Gets left operand for binary operators (like **+**)
    - Leftmost object must be of same class as operator
  - Non member functions
    - Need parameters for both operands
    - Can have object of different class than operator
    - Must be a **`friend`** to access **`private`** or **`protected`** data
  - Operator member function called when
    - Left operand of binary operator of same class
    - Single operand of unitary operator of same class

Operator Functions As Class Members Vs. As Friend Functions

8

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

Operator Functions As Class Members Vs. As Friend Functions

9

- 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 non-member overload function
    - **Long int + HugeIntClass**

Overloading Stream-Insertion and Stream-Extraction Operators

10

- **<<** and **>>**
  - Already overloaded to process each built-in type
  - Can also process a user-defined class
- Example program
  - Class **PhoneNumber**
    - Holds a telephone number
  - Print out formatted number automatically
    - **(123) 456-7890**

11

Outline

fig08_03.cpp
(1 of 3)

```
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 // PhoneNumber class definitio
17 class PhoneNumber {
18    friend ostream &operator<<(
19    friend istream &operator>>(
20
21 private:
22    char areaCode[ 4 ];  // 3-d
23    char exchange[ 4 ];  // 3-d
24    char line[ 5 ];      // 4-digit line and null
25
26 }; // end class PhoneNumber
```

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

They must be non-member **friend** functions, since the object of class **Phonenumber** appears on the right of the operator.

cin << object
cout >> object

12

Outline

fig08_03.cpp
(2 of 3)

```
27
28 // overloaded stream-insertion operator; cannot be
29 // a member function if we would like to invoke it with
30 // cout << somePhoneNumber;
31 ostream &operator<<( ostream &output, const PhoneNumber &num )
32 {
33    output << "(" << num.areaCode << ") "
34           << num.exchange << "-" << num.line;
35
36    return output;     // enables cout << a << b << c;
37
38 } // end function operator<<
39
40 // overloaded stream-extraction operator; cannot be
41 // a member function if we would like to invoke it with
42 // cin >> somePhoneNumber;
43 istream &operator>>( istream
44 {
45    input.ignore();                    // skip (
46    input >> setw( 4 ) >> num.areaCode; // input are
47    input.ignore( 2 );                 // skip ) an
48    input >> setw( 4 ) >> num.exchange;
49    input.ignore();
50    input >> setw( 5 ) >> num.line;
51
52    return input;      // enables cin >
```

The expression:
cout << phone;
is interpreted as the function call:
operator<<(cout, phone);

output is an alias for **cout.**

This allows objects to be cascaded.
<< phone1 << phone2;
alls
ator<<(cout, phone1), and
eturns **cout**.

Next, **cout << phone2** executes.

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

Stream manipulator **setw** restricts number of characters read. **setw(4)** allows 3 characters to be read, leaving room for the null character.

Outline

fig08_03.cpp
(3 of 3)

fig08_03.cpp
output (1 of 1)

```
53
54 } // end function operator>>
55
56 int main()
57 {
58    PhoneNumber phone; // create object phone
59
60    cout << "Enter phone number in the form (123) 456-7890:\n";
61
62    // cin >> phone invokes operator>> by implicitly issuing
63    // the non-member function call operator>>( cin, phone )
64    cin >> phone;
65
66    cout << "The phone number entered was: " ;
67
68    // cout << phone invokes operator<< by implicitly issuing
69    // the non-member function call operator<<( cout, phone )
70    cout << phone << endl;
71
72    return 0;
73
74 } // end main
```

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


Overloading Unary Operators

- Overloading unary operators
  - Non-**`static`** member function, no arguments
  - Non-member function, one argument
    - Argument must be class object or reference to class object
  - Remember, **`static`** functions only access **`static`** data

Overloading Unary Operators

- Upcoming example (8.10)
  - Overload **`!`** to test for empty string
  - If non-**`static`** member function, needs no arguments
    - **`!s`** becomes **`s.operator!()`**

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

  - If non-member function, needs one argument
    - **`s!`** becomes **`operator!(s)`**

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


Overloading Binary Operators

- Overloading binary operators
  - Non-**`static`** member function, one argument
  - Non-member function, two arguments
    - One argument must be class object or reference
- Upcoming example
  - If non-**`static`** member function, needs one argument

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

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

Overloading Binary Operators

17

- Upcoming example
  - If non-member function, needs two arguments
  - Example:

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

  - **y += z** equivalent to **operator+=(y, z)**

Case Study: Array class

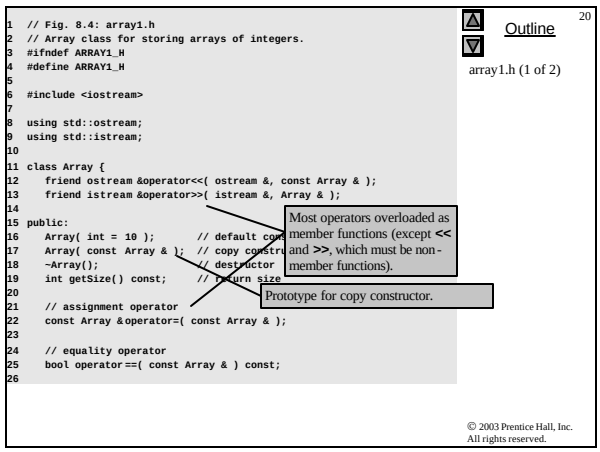
18

- Arrays in C++
  - No range checking
  - Cannot be compared meaningfully with **==**
  - No array assignment (array names **const** pointers)
  - Cannot input/output entire arrays at once
    - One element at a time
- Example:Implement an **Array** class with
  - Range checking
  - Array assignment
  - Arrays that know their size
  - Outputting/inputting entire arrays with **<<** and **>>**
  - Array comparisons with **==** and **!=**

Case Study: Array class

19

- Copy constructor
  - Used whenever copy of object needed
    - Passing by value (return value or parameter)
    - Initializing an object with a copy of another
      - **Array newArray(oldArray);**
      - **newArray** copy of **oldArray**
  - Prototype for class **Array**
    - **Array(const Array &);**
    - *Must* take reference
      - Otherwise, pass by value
      - Tries to make copy by calling copy constructor…
      - Infinite loop




```
1  // Fig. 8.4: array1.h
2  // Array class for storing arrays of 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
15 public:
16    Array( int = 10 );          // default con
17    Array( const Array & );     // copy constru
18    ~Array();                   // destructor
19    int getSize() const;        // return size
20
21    // assignment operator
22    const Array &operator=( const Array & );
23
24    // equality operator
25    bool operator==( const Array & ) const;
26
```

Outline

array1.h (2 of 2)

```
27    // inequality operator; returns opposite of == operator
28    bool operator!=( const Array &right ) const
29    {
30       return ! ( *this == right ); // invokes Array::operator==
31
32    } // end function operator!=
33
34    // subscript operator for non-con
35    int &operator[]( int );
36
37    // subscript operator for const objects returns rvalue
38    const int &operator[]( int ) const;
39
40 private:
41    int size; // array size
42    int *ptr; // pointer to first element of array
43
44 }; // end class Array
45
46 #endif
```

!= operator simply returns opposite of **==** operator. Thus, only need to define the **==** operator.

Outline

array1.cpp (1 of 7)

```
1  // Fig 8.5: array1.cpp
2  // Member function definitions for class Array
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 <new>       // C++ standard "new" operator
14
15 #include <cstdlib>   // exit function prototype
16
17 #include "array1.h"  // Array class definition
18
19 // default constructor for class Array (default size 10)
20 Array::Array( int arraySize )
21 {
22    // validate arraySize
23    size = ( arraySize > 0 ? arraySize : 10 );
24
25    ptr = new int[ size ]; // create space for array
26
```


Outline

array1.cpp (2 of 7)

```
27    for ( int i = 0; i < size; i++ )
28       ptr[ i ] = 0;          // initialize array
29
30 } // end Array default constructor
31
32 // copy constructor for class Array;
33 // must receive a reference to prev
34 Array::Array( const Array &arrayTo
35    : size( arrayToCopy.size )
36 {
37    ptr = new int[ size ]; // create space for array
38
39    for ( int i = 0; i < size; i++ )
40       ptr[ i ] = arrayToCopy.ptr[ i ];  // copy into object
41
42 } // end Array copy constructor
43
44 // destructor for class Array
45 Array::~Array()
46 {
47    delete [] ptr;  // reclaim array space
48
49 } // end destructor
50
```

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

Outline

array1.cpp (3 of 7)

```
51 // return size of array
52 int Array::getSize() const
53 {
54    return size;
55
56 } // end function getSize
57
58 // overloaded assignment operator;
59 // const return avoids: ( a1 = a2 ) = a3
60 const Array &Array::operator=( const Array &right )
61 {
62    if ( &right != this ) {  // check for self-assignment
63
64       // for arrays of different sizes, deallocate original
65       // left-side array, then allocate new left-side array
66       if ( size != right.size ) {
67          delete [] ptr;         // reclaim space
68          size = right.size;     // resize this object
69          ptr = new int[ size ]; // create space for array copy
70
71       } // end inner if
72
73       for ( int i = 0; i < size; i++ )
74          ptr[ i ] = right.ptr[ i ];  // copy array into object
75
76    } // end outer if
```

Want to avoid self-assignment.

array1.cpp (4 of 7)

```
77
78    return *this;   // enables x = y = z, for example
79
80 } // end function operator=
81
82 // determine if two arrays are equal and
83 // return true, otherwise return false
84 bool Array::operator==( const Array &right ) const
85 {
86    if ( size != right.size )
87       return false;    // arrays of different sizes
88
89    for ( int i = 0; i < size; i++ )
90
91       if ( ptr[ i ] != right.ptr[ i ] )
92          return false; // arrays are not equal
93
94    return true;        // arrays are equal
95
96 } // end function operator==
97
```


array1.cpp (5 of 7)

```
98 // overloaded subscript operator for non-const Arrays
99 // reference return creates an lvalue
100 int &Array::operator[]( int subscript )
101 {
102    // check for subscript out of range err
103    if ( subscript < 0 || subscript >= size
104       cout << "\nError: Subscript " << subs
105            << " out of range" << endl;
106
107       exit( 1 );  // terminate program; subscript out of range
108
109    } // end if
110
111    return ptr[ subscript ]; // reference return
112
113 } // end function operator[]
114
```

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

exit() (header **<cstdlib>**) ends the program.

array1.cpp (6 of 7)

```
115 // overloaded subscript operator for const Arrays
116 // const reference return creates an rvalue
117 const int &Array::operator[]( int subscript ) const
118 {
119    // check for subscript out of range error
120    if ( subscript < 0 || subscript >= size ) {
121       cout << "\nError: Subscript " << subscript
122            << " out of range" << endl;
123
124       exit( 1 );  // terminate program; subscript out of range
125
126    } // end if
127
128    return ptr[ subscript ]; // const reference return
129
130 } // end function operator[]
131
132 // overloaded input operator for class Array;
133 // inputs values for entire array
134 istream &operator>>( istream &input, Array &a )
135 {
136    for ( int i = 0; i < a.size; i++ )
137       input >> a.ptr[ i ];
138
139    return input;   // enables cin >> x >> y;
140
141 } // end function
```


array1.cpp (7 of 7)

```
142
143 // overloaded output operator for class Array
144 ostream &operator<<( ostream &output, const Array &a )
145 {
146    int i;
147
148    // output private ptr-based array
149    for ( i = 0; i < a.size; i++ ) {
150       output << setw( 12 ) << a.ptr[ i ];
151
152       if ( ( i + 1 ) % 4 == 0 ) // 4 numbers per row of output
153          output << endl;
154
155    } // end for
156
157    if ( i % 4 != 0 )  // end last line of output
158       output << endl;
159
160    return output;   // enables cout << x << y;
161
162 } // end function operator<<
```

Outline

fig08_06.cpp
(1 of 3)

```
1  // Fig. 8.6: fig08_06.cpp
2  // Array class test program.
3  #include <iostream>
4
5  using std::cout;
6  using std::cin;
7  using std::endl;
8
9  #include "array1.h"
10
11 int main()
12 {
13    Array integers1( 7 );  // seven-element Array
14    Array integers2;       // 10-element Array by default
15
16    // print integers1 size and contents
17    cout << "Size of array integers1 is "
18         << integers1.getSize()
19         << "\nArray after initialization:\n" << integers1;
20
21    // print integers2 size and contents
22    cout << "\nSize of array integers2 is "
23         << integers2.getSize()
24         << "\nArray after initialization:\n" << integers2;
25
```


Outline

fig08_06.cpp
(2 of 3)

```
26    // input and print integers1 and integers2
27    cout << "\nInput 17 integers:\n";
28    cin >> integers1 >> integers2;
29
30    cout << "\nAfter input, the arrays contain:\n"
31         << "integers1:\n" << integers1
32         << "integers2:\n" << integers2;
33
34    // use overloaded inequality (!=) operator
35    cout << "\nEvaluating: integers1 != integers2\n";
36
37    if ( integers1 != integers2 )
38       cout << "integers1 and integers2 are not equal\n";
39
40    // create array integers3 using integers1 as an
41    // initializer; print size and contents
42    Array integers3( integers1 );  // calls copy constructor
43
44    cout << "\nSize of array integers3 is "
45         << integers3.getSize()
46         << "\nArray after initialization:\n" << integers3;
47
```


Outline

fig08_06.cpp
(3 of 3)

```
48    // use overloaded assignment (=) operator
49    cout << "\nAssigning integers2 to integers1:\n";
50    integers1 = integers2;  // note target is smaller
51
52    cout << "integers1:\n" << integers1
53         << "integers2:\n" << integers2;
54
55    // use overloaded equality (==) operator
56    cout << "\nEvaluating: integers1 == integers2\n";
57
58    if ( integers1 == integers2 )
59       cout << "integers1 and integers2 are equal\n";
60
61    // use overloaded subscript operator to create rvalue
62    cout << "\nintegers1[5] is " << integers1[ 5 ];
63
64    // use overloaded subscript operator to create lvalue
65    cout << "\n\nAssigning 1000 to integers1[5]\n";
66    integers1[ 5 ] = 1000;
67    cout << "integers1:\n" << integers1;
68
69    // attempt to use out-of-range subscript
70    cout << "\nAttempt to assign 1000 to integers1[15]" << endl;
71    integers1[ 15 ] = 1000;  // ERROR: out of range
72
73    return 0;
74
75 } // end main
```


Outline

fig08_06.cpp
output (1 of 3)

```
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
```

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

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

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

Evaluating: integers1 == integers2
integers1 and integers2 are equal

integers1[5] is 13
```


fig08_06.cpp
output (2 of 3)


```
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
```


fig08_06.cpp
output (3 of 3)

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
    - 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

Converting between Types

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

Converting between Types

- 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 **`<<`**
  - Compiler can only do 1 cast

Case Study: A `String` Class

- Build class **`String`**
  - 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

Outline

string1.h (1 of 3)

```
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 construc
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.

Outline

string1.h (2 of 3)

```
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>=
```

Outline

string1.h (3 of 3)

```
54
55    char &operator[]( int );                  // s
56    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
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

string1.cpp (1 of 8)

```
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

string1.cpp (2 of 8)

```
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

string1.cpp (3 of 8)

```
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

string1.cpp (4 of 8)

```
// is this String empty?
bool String::operator!() const
{
   return length == 0;

} // end function operator!

// is this String equal to right String?
bool String::operator==( const String &right ) const
{
   return strcmp( sPtr, right.sPtr ) == 0;

} // end function operator==

// is this String less than right String?
bool String::operator<( const String &right ) const
{
   return strcmp( sPtr, right.sPtr ) < 0;

} // end function operator<

```


Outline

string1.cpp (5 of 8)

```
// return reference to character in String as lvalue
char &String::operator[]( int subscript )
{
   // test for subscript out of range
   if ( subscript < 0 || subscript >= length ) {
      cout << "Error: Subscript " << subscript
           << " out of range" << endl;

      exit( 1 );  // terminate program
   }

   return sPtr[ subscript ];  // creates lvalue

} // end function operator[]

// return reference to character in String as rvalue
const char &String::operator[]( int subscript ) const
{
   // test for subscript out of range
   if ( subscript < 0 || subscript >= length ) {
      cout << "Error: Subscript " << subscript
           << " out of range" << endl;

      exit( 1 );  // terminate program
   }

   return sPtr[ subscript ];  // creates rvalue

} // end function operator[]
```


Outline

string1.cpp (6 of 8)

```

// return a substring beginning at index and
// of length subLength
String String::operator()( int index, int subLength )
{
   // if index is out of range or substring length < 0,
   // return an empty String object
   if ( index < 0 || index >= length || subLength < 0 )
      return "";  // converted to a String object automatically

   // determine length of substring
   int len;

   if ( ( subLength == 0 ) || ( index + subLength > length ) )
      len = length - index;
   else
      len = subLength;

   // allocate temporary array for substring and
   // terminating null character
   char *tempPtr = new char[ len + 1 ];

   // copy substring into char array and terminate string
   strncpy( tempPtr, &sPtr[ index ], len );
   tempPtr[ len ] = '\0';
```


Outline

string1.cpp (7 of 8)

```

   // create temporary String object containing the substring
   String tempString( tempPtr );
   delete [] tempPtr;  // delete temporary array

   return tempString;  // return copy of the temporary String

} // end function operator()

// return string length
int String::getLength() const
{
   return length;

} // end function getLenth

// utility function called by constructors and operator=
void String::setString( const char *string2 )
{
   sPtr = new char[ length + 1 ]; // allocate memory
   strcpy( sPtr, string2 );       // copy literal to object

} // end function setString
```

```
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

string1.cpp (8 of 8)


```
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 << '\"'
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

fig08_09.cpp
(1 of 4)


```
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 << "\"";
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

fig08_09.cpp
(2 of 4)


```
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

fig08_09.cpp
(3 of 4)

```
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
```


fig08_09.cpp
(4 of 4)


```
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"`**).

fig08_09.cpp
(1 of 3)


```
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
```


fig08_09.cpp
(2 of 3)


```
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
```


fig08_09.cpp
(3 of 3)

Overloading ++ and - -

57

- 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 )`**

Overloading ++ and - -

58

- 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++( Data &, int );`**
    - **`d1++`** same as **`operator++( d1, 0 )`**
  - Integer parameter does not have a name
    - Not even in function definition

8.11 Overloading ++ and - -

59

- 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

Case Study: A `Date` Class

60

- 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

```
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 = 
14    void setDate( int, int, int ); // se
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.

date1.h (1 of 2)


```
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
```


date1.h (2 of 2)


```
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
```


date1.cpp (1 of 5)


```
24    // test for a leap year
25    if ( month == 2 && leapYear( year ) )
26       day = ( dd >= 1 && dd <= 29 ) ? dd : 1;
27    else
28       day = ( dd >= 1 && dd <= days[ month ] ) ? dd : 1;
29
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
41 // overloaded postincrement operator; no
42 // integer parameter does not have a par
43 Date Date::operator++( int )
44 {
45    Date temp = *this;  // hold current s
46    helpIncrement();
47
48    // return unincremented, saved, tempo
49    return temp;   // value return; not a
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.

date1.cpp (2 of 5)

```
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
```


date1.cpp (3 of 5)


```
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
```


date1.cpp (4 of 5)


```
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<<
```


date1.cpp (5 of 5)


```
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 << "\n\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 );
```


fig08_12.cpp
(1 of 2)

```
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

fig08_12.cpp
(2 of 2)


```
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

fig08_12.cpp
output (1 of 1)

Standard Library Classes `string` and `vector`

- 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`**

8.13 Standard Library Classes `string` and `vector`

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

Standard Library Classes `string` and `vector`

- Class **`string`**
  - 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)

Outline

fig08_13.cpp
(1 of 4)

```
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 << '\"'
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

fig08_13.cpp
(2 of 4)

```
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 << "\"";
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

fig08_13.cpp
(3 of 4)

```
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
```

```
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

fig08_13.cpp
(4 of 4)

Standard Library Classes `string` and `vector`

- 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


```
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

fig08_14.cpp
(1 of 5)


```
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

fig08_14.cpp
(2 of 5)

```
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

fig08_14.cpp
(3 of 5)


```
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 << "\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

fig08_14.cpp
(4 of 5)


```
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
106    } // end for
107
108    if ( i % 4 != 0 )
109       cout << endl;
110
111 } // end function outputVector
112
113 // input vector contents
114 void inputVector( vector< int > &array )
115 {
116    for ( int i = 0; i < array.size(); i++ )
117       cin >> array[ i ];
118
119 } // end function inputVector
```


Outline

fig08_14.cpp
(5 of 5)


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

fig08_14.cpp
output (1 of 2)

```
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

fig08_14.cpp
output (2 of 2)