W8.2 Operator Overloading

- Fundamentals of Operator Overloading
- Restrictions on Operator Overloading
- Operator Functions as Class Members vs. as friend Functions
- Overloading Stream Insertion and Extraction Operators
- Overloading Unary Operators
- Overloading Binary Operators
- Case Study: An Array Class
- Converting between Types
- Case Study: A String Class
- Overloading ++ and --
- Case Study: A Date Class
• 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
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
Restrictions on Operator Overloading

- C++ operators that can be overloaded

<table>
<thead>
<tr>
<th>Operators that can be overloaded</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
</tr>
<tr>
<td>~</td>
</tr>
<tr>
<td>/=</td>
</tr>
<tr>
<td>&lt;&lt;=</td>
</tr>
<tr>
<td>--</td>
</tr>
<tr>
<td>new[]</td>
</tr>
</tbody>
</table>

- C++ Operators that cannot be overloaded

<table>
<thead>
<tr>
<th>Operators that cannot be overloaded</th>
</tr>
</thead>
<tbody>
<tr>
<td>.</td>
</tr>
</tbody>
</table>
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
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 friends if needs to access private or protected members
  – Enable the operator to be commutative (a+b, b+a)
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
// Fig. 8.3: fig08_03.cpp
// Overloading the stream-insertion and
// stream-extraction operators.
#include <iostream>

using std::cout;
using std::cin;
using std::endl;
using std::ostream;
using std::istream;

#include <iomanip>

using std::setw;

class PhoneNumber {
    friend ostream &operator<<( ostream&, const PhoneNumber & );
    friend istream &operator>>( istream&, PhoneNumber & );

private:
    char areaCode[ 4 ]; // 3-digit area code and null
    char exchange[ 4 ]; // 3-digit exchange and null
    char line[ 5 ]; // 4-digit line and null
};

// Overloaded stream-insertion operator (cannot be
// a member function if we would like to invoke it with
// cout << somePhoneNumber;).
ostream &operator<<( ostringstream &output, const PhoneNumber &num )
{
The function call

\texttt{cin >> phone;}

interpreted as

\texttt{operator>>(cin, phone);}

\texttt{input} is an alias for \texttt{cin}, and \texttt{num} is an alias for \texttt{phone}. 
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
  - 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:
    ```cpp
    class String {
    public:
      bool operator!() const;
      ... 
    };
    ```
Overloading Unary Operators

– Example declaration as a non-member function

```cpp
class String {
    friend bool operator!( const String & )
    ...
}
```
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 \( y.operator+=(( z ) ) \)
Overloading Binary Operators

- Non-member function, two arguments
- Example:

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

- \( y += z \) is equivalent to `operator+=( y, z )`
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 `!=`
// Fig. 8.4: array1.h
// Simple class Array (for integers)
#ifndef ARRAY1_H
#define ARRAY1_H

#include <iostream>
using std::ostream;
using std::istream;

class Array {
    friend ostream &operator<<( ostream &, const Array & );
    friend istream &operator>>( istream &, Array & );

public:
    Array( int = 10 );                   // default constructor
    Array( const Array & );              // copy constructor
    ~Array();                            // destructor
    int getSize() const;                 // return size
    const Array &operator=( const Array & ); // assign arrays
    bool operator==( const Array & ) const;  // compare equal

    bool operator!=( const Array &right ) const { return ! ( *this == right ); }

    int &operator[]( int );              // subscript operator
    const int &operator[]( int ) const;  // subscript operator
    static int getArrayCount();          // Return count of
                                            // arrays instantiated.

private:
    int size; // size of the array
    int *ptr; // pointer to first element of array
    static int arrayCount; // # of Arrays instantiated.
```cpp
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
```
C++ code for Array class:

```cpp
// Copy constructor for class Array
Array::Array( const Array &init ) : size( init.size ) {
    ptr = new int[ size ]; // create space for array
    assert( ptr != 0 );    // terminate if memory not allocated
    ++arrayCount;          // count one more object

    for ( int i = 0; i < size; i++ )
        ptr[ i ] = init.ptr[ i ]; // copy init into object
}

// Destructor for class Array
Array::~Array() {
    delete [] ptr;            // reclaim space for array
    --arrayCount;             // one fewer object
}

// Get the size of the array
int Array::getSize() const { return size; }

// Overloaded assignment operator
const Array &Array::operator=( const Array &right ) {
    if ( &right != this ) { // check for self-assignment
        // for arrays of different sizes, deallocate original
        // left side array, then allocate new left side array.
        if ( size != right.size ) {
            delete [] ptr;    // reclaim space
            } // check for self-assignment
```
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 // Determine if two arrays are equal and
113 // return true, otherwise return false.
114 bool Array::operator==( const Array &right ) const
115{
116    if ( size != right.size )
117        return false;    // arrays of different sizes
118
119    for ( int i = 0; i < size; i++ )
120        if ( ptr[ i ] != right.ptr[ i ] )
121            return false; // arrays are not equal
122
123    return true;        // arrays are equal
124}
125
126 // Overloaded subscript operator for non-const Arrays
127 // reference return creates an lvalue
128 int &Array::operator[]( int subscript )
129{
130    // check for subscript out of range error
131    assert( 0 <= subscript && subscript < size );
return ptr[subscript]; // reference return

// Overloaded subscript operator for const Arrays
// const reference return creates an rvalue
const int &Array::operator[](int subscript) const
{
  // check for subscript out of range error
  assert(0 <= subscript && subscript < size);
  return ptr[subscript]; // const reference return
}

// Return the number of Array objects instantiated
// static functions cannot be const
int Array::getArrayCount()
{
  return arrayCount;
}

// Overloaded input operator for class Array;
// inputs values for entire array.
istream &operator>>(istream &input, Array &a)
{
  for (int i = 0; i < a.size; i++)
    input >> a.ptr[i];
  return input; // enables cin >> x >> y;
}

// Overloaded output operator for class Array
ostream &operator<<(ostream &output, const Array &a)
{
```cpp
int i;

for ( i = 0; i < a.size; i++ ) {
    output << setw( 12 ) << a.ptr[i];

    if ( ( i + 1 ) % 4 == 0 ) // 4 numbers per row of output
        output << endl;
}

if ( i % 4 != 0 )
    output << endl;

return output;   // enables cout << x << y;
}

#include <iostream>

using std::cout;
using std::cin;
using std::endl;

#include "array1.h"

int main()
{
    // no objects yet
    cout << "# of arrays instantiated = "
        << Array::getArrayCount() << '\n';
```
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"
   << ( integers1 != integers2 )
   << "They are not equal\n";

// create array integers3 using integers1 as an
// initializer; print size and contents

Array integers3( integers1 );
cout << "\nSize of array integers3 is " << integers3.getSize() << "\nArray after initialization:
" << integers3 << '\n';

// use overloaded assignment (=) operator
cout << "Assigning integers2 to integers1:
"; integers1 = integers2;
cout << "integers1:\n" << integers1 << 'n';

// use overloaded equality (==) operator
cout << "Evaluating: integers1 == integers2
"; if ( integers1 == integers2 )
  cout << "They are equal\n\n";

// use overloaded subscript operator to create rvalue

// use overloaded subscript operator to create lvalue
cout << "Assigning 1000 to integers1[5]
"; integers1[5] = 1000;
cout << "integers1:\n" << integers1 << 'n';

// attempt to use out of range subscript
cout << "Attempt to assign 1000 to integers1[15]"
integers1[15] = 1000; // ERROR: out of range

return 0;
# of arrays instantiated = 0
# of arrays instantiated = 2

Size of array integers1 is 7
Array after initialization:

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Size of array integers2 is 10
Array after initialization:

<p>| | | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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:

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>6</td>
<td>7</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

integers2:

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>17</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Evaluating: integers1 != integers2
They are not equal

Size of array integers3 is 7
Array after initialization:

<p>| | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>6</td>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Assigning integers2 to integers1:

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>17</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

integers2:

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>17</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Evaluating: integers1 == integers2
They are equal

integers1[5] is 13
Assigning 1000 to integers1[5]

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>17</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Attempt to assign 1000 to integers1[15]
Assertion failed: 0 <= subscript && subscript < size, file Array1.cpp, line 95 abnormal program termination
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
Converting between Types

\[ \text{A::operator int()} \text{ const;} \]
• Declares an overloaded cast operator function for converting an object of \text{A} into an integer

\[ \text{A::operator otherClass()} \text{ const;} \]
• Declares an overloaded cast operator function for converting an object of \text{A} into an object of \text{otherClass}

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

\[ \text{cout} \ll \text{s;} \]

The compiler calls the overloaded cast operator function \text{operator char *} to convert the object into a \text{char *} and uses the resulting \text{char *} in the expression
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
// Fig. 8.5: string1.h
// Definition of a String class
#ifndef STRING1_H
#define STRING1_H

#include <iostream>

using std::ostream;
using std::istream;

class String {
    friend ostream &operator<<( ostream &, const String & );
    friend istream &operator>>( istream &, String & );

public:
    String( const char * = "" ); // conversion/default ctor
    String( const String & );    // copy constructor
    ~String();                   // destructor
    const String &operator=( const String & );  // assignment
    const String &operator+=( const String & ); // concatenation
    bool operator!() const;                  // is String empty?
    bool operator==( const String & ) const; // test s1 == s2
    bool operator<( const String & ) const;  // test s1 < s2
    bool operator!=( const String & right ) const
        { return !( *this == right ); }
    bool operator>=( const String & ) const; // test s1 >= s2
    bool operator>( const String &right ) const
        { return right < *this; }
    // test s1 <= s2
};

// Fig. 8.5: string1.h
bool operator<=( const String &right ) const
    { return !( right < *this ); }  

// test s1 >= s2
bool operator>=( const String &right ) const
    { return !( *this < right ); }  

char &operator[]( int );             // subscript operator
const char &operator[]( int ) const; // subscript operator
String operator()( int, int );       // return a substring
int getLength() const;               // return string length

private:
    int length;                   // string length
    char *sPtr;                   // pointer to start of string

    void setString( const char * ); // utility function

#endif // Fig. 8.5: string1.cpp
// Member function definitions for class String
#include <iostream>
#include <iomanip>
using std::cout;
using std::endl;
using std::setw;
#include <iomanip>
using std::cout;
using std::endl;
using std::setw;
// Conversion constructor: Convert char * to String
String::String( const char *s ) : length( strlen( s ) )
{
    cout << "Conversion constructor: " << s << '\n';
    setString( s ); // call utility function
}

// Copy constructor
String::String( const String &copy ) : length( copy.length )
{
    cout << "Copy constructor: " << copy.sPtr << '\n';
    setString( copy.sPtr ); // call utility function
}

// Destructor
String::~String()
{
    cout << "Destructor: " << sPtr << '\n';
    delete [] sPtr; // reclaim string
}

// Overloaded = operator; avoids self assignment
const String &String::operator=( const String &right )
{
    cout << "operator= called\n";
    if ( &right != this ) {
        // avoid self assignment

delete [] sPtr;        // prevents memory leak
length = right.length; // new String length
setString( right.sPtr ); // call utility function
}
else
    cout << "Attempted assignment of a String to itself\n";
return *this;  // enables cascaded assignments
}

// Concatenate right operand to this object and
// store in this object.
const String &String::operator+=( const String &right )
{
    char *tempPtr = sPtr;        // hold to be able to delete
    length += right.length;      // new String length
    sPtr = new char[ length + 1 ]; // create space
    assert( sPtr != 0 );   // terminate if memory not allocated
    strcpy( sPtr, tempPtr );     // left part of new String
    strcat( sPtr, right.sPtr );  // right part of new String
    delete [] tempPtr;           // reclaim old space
    return *this;                // enables cascaded calls
}

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

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

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

// Return a reference to a character in a String as an lvalue.
char &String::operator[]( int subscript )
{
    // First test for subscript out of range
    assert( subscript >= 0 && subscript < length );
    return sPtr[ subscript ]; // creates lvalue
}

// Return a reference to a character in a String as an rvalue.
const char &String::operator[]( int subscript ) const
{
    // First test for subscript out of range
    assert( subscript >= 0 && subscript < length );
    return sPtr[ subscript ]; // creates rvalue
}

// Return a substring beginning at index and
// of length subLength
String String::operator()( int index, int subLength )
{
    // ensure index is in range and substring length >= 0
    assert( index >= 0 && index < length && subLength >= 0 );

    // 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 ];
assert( tempPtr != 0 ); // ensure space allocated

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

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

return tempString; // return copy of the temporary String

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

// Utility function to be called by constructors and
// assignment operator.
void String::setString( const char *string2 )
{
    sPtr = new char[ length + 1 ]; // allocate storage
    assert( sPtr != 0 ); // terminate if memory not allocated
    strcpy( sPtr, string2 ); // copy literal to object
// Overloaded output operator
ostream &operator<<( ostream &output, const String &s )
{
    output << s.sPtr;
    return output;   // enables cascading
}

// Overloaded input operator
istream &operator>>( istream &input, String &s )
{
    char temp[ 100 ];   // buffer to store input
    input >> setw( 100 ) >> temp;
    s = temp;        // use String class assignment operator
    return input;    // enables cascading
}

// Fig. 8.5: fig08_05.cpp
// Driver for class String
#include <iostream>

using std::cout;
using std::endl;

#include "string1.h"

int main()
{
    String s1( "happy" ), s2( "birthday" ), s3;
    Conversion constructor: happy
    Conversion constructor: birthday
    Conversion constructor:
2. Function calls

```cpp
// test overloaded equality and relational operators
cout << "s1 is " \"" \<< s1 \"\" \<< s2
<< \"\"; s3 is \"" \<< s3 \"\";
\n\nThe results of comparing s2 and s1:
\n\ns2 == s1 yields "false"
\ns2 != s1 yields "true"
\ns2 > s1 yields "false"
\ns2 < s1 yields "true"
\ns2 >= s1 yields "false"
\ns2 <= s1 yields "true";

// test overloaded String empty (!) operator
if ( !s3 ) {
    cout << "s3 is empty; assigning s1 to s3; \n"
    s3 = s1; // test overloaded assignment
    cout << "s3 is \"" \<< s3 \"\";\n}

// test overloaded String concatenation operator
s1 += s2 yields s1 = "happy birthday"

// test conversion constructor
s1 += \" to you\" yields
Conversion constructor: \" to you
Destructor: \" to you

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\"
```
```cpp
cout << "s1 = " << s1 << "\n\n";  // s1 = happy birthday to you

// test overloaded function call operator () for substring
cout << "The substring of s1 starting at\n" << "location 0 for 14 characters, s1(0, 14), is:\n" << s1(0, 14) << "\n\n";

// test substring "to-end-of-String" option
cout << "The substring of s1 starting at\n" << "location 15, s1(15, 0), is: " << s1(15, 0) << "\n\n";  // 0 is "to end of string"

// test copy constructor
String *s4Ptr = new String(s1);
cout << "*s4Ptr = " << *s4Ptr << "\n\n";

// test assignment (=) operator with self-assignment
cout << "assigning *s4Ptr to *s4Ptr\n";
*s4Ptr = *s4Ptr;  // test overloaded assignment
cout << "*s4Ptr = " << *s4Ptr << '\n';

// test destructor
delete s4Ptr;

// test using subscript operator to create lvalue
s1[0] = 'H';
s1[6] = 'B';
cout << "\nsl after s1[0] = 'H' and s1[6] = 'B' is: " << s1 << "\n\n";

// 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:

// Conversion constructor: to you
// Copy constructor: to you
// Destructor: to you
The substring of s1 starting at
location 0 for 14 characters, s1(0, 14), is:

// *s4Ptr = happy birthday to you
// 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
```

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 }

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

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 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
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
      \[
      \text{d1.operator++();} \quad // \text{for ++d1}
      \]
    • convention adopted that when compiler sees postincrementing expression, it will generate the member-function call
      \[
      \text{d1.operator++( 0 );} \quad // \text{for d1++}
      \]
    • 0 is a dummy value to make the argument list of \text{operator++} distinguishable from the argument list for \text{++operator}
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
class Date {
    friend ostream &operator<<( ostream &, const Date & );

public:
    Date( int m = 1, int d = 1, int y = 1900 ); // constructor
    void setDate( int, int, int ); // set the date

    Date &operator++();            // preincrement operator
    Date operator++( int );        // postincrement operator
    const Date &operator+=( int ); // add days, modify object
    bool leapYear( int ) const;    // is this a leap year?
    bool endOfMonth( int ) const;  // is this end of month?

private:
    int month;
    int day;
    int year;

    static const int days[];       // array of days per month

    void helpIncrement();          // utility function

};

#ifndef DATE1_H
#define DATE1_H
#include <iostream>

using std::ostream;

class Date {
    friend ostream &operator<<( ostream &, const Date & );

public:
    Date( int m = 1, int d = 1, int y = 1900 ); // constructor
    void setDate( int, int, int ); // set the date

    Date &operator++();            // preincrement operator
    Date operator++( int );        // postincrement operator
    const Date &operator+=( int ); // add days, modify object
    bool leapYear( int ) const;    // is this a leap year?
    bool endOfMonth( int ) const;  // is this end of month?

private:
    int month;
    int day;
    int year;

    static const int days[];       // array of days per month

    void helpIncrement();          // utility function

};

#endif
// Fig. 8.6: date1.cpp
// Member function definitions for Date class
#include <iostream>
#include "date1.h"

// Initialize static member at file scope;
// one class-wide copy.
const int Date::days[] = { 0, 31, 28, 31, 30, 31, 30,

// Date constructor
Date::Date( int m, int d, int y ) { setDate( m, d, y ); }  

// Set the date
void Date::setDate( int mm, int dd, int yy )
{
    month = ( mm >= 1 && mm <= 12 ) ? mm : 1;
    year = ( yy >= 1900 && yy <= 2100 ) ? yy : 1900;
    // test for a leap year
    if ( month == 2 && leapYear( year ) )
        day = ( dd >= 1 && dd <= 29 ) ? dd : 1;
    else
        day = ( dd >= 1 && dd <= days[ month ] ) ? dd : 1;
}

// Preincrement operator overloaded as a member function.
Date &Date::operator++()
{
    helpIncrement();  
    return *this;  // reference return to create an lvalue
}
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 {
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
105 void 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
121 ostream &operator<<( ostream &output, const Date &d )
122 {
124                                           "July", "August", "September", "October",
125                                           "November", "December" };
126    output << monthName[ d.month ] << ' ' << d.day << ", " << d.year;
127    return output;  // enables cascading
128 }
// Fig. 8.6: fig08_06.cpp
// Driver for class Date
#include <iostream>

using std::cout;
using std::endl;

#include "date1.h"

int main()
{
    Date d1, d2( 12, 27, 1992 ), d3( 0, 99, 8045 );
    cout << "d1 is " << d1
         << "\nd2 is " << d2
         << "\n\nd3 is " << d3 << "\n";

    cout << "d2 += 7 is " << ( d2 += 7 ) << "\n";

    d3.setDate( 2, 28, 1992 );
    cout << "  d3 is " << d3;

    ++d3;
    cout << "++d3 is " << ++d3; // d3 is February 29, 1992

    Date d4( 3, 18, 1969 );
    cout << "Testing the preincrement operator:
         " << d4 << "\n";
    ++d4;
    cout << "++d4 is " << ++d4; // d4 is March 19, 1969

    return 0;
}
d1 is January 1, 1900

d2 is December 27, 1992

d3 is January 1, 1900

\[ d2 += 7 \text{ is January 3, 1993} \]

\[ d3 \text{ is February 28, 1992} \]
\[ ++d3 \text{ is February 29, 1992} \]

Testing the preincrement operator:

\[ d4 \text{ is March 18, 1969} \]
\[ ++d4 \text{ is March 19, 1969} \]
\[ d4 \text{ is March 19, 1969} \]

Testing the postincrement operator:

\[ d4 \text{ is March 19, 1969} \]
\[ d4++ \text{ is March 19, 1969} \]
\[ d4 \text{ is March 20, 1969} \]