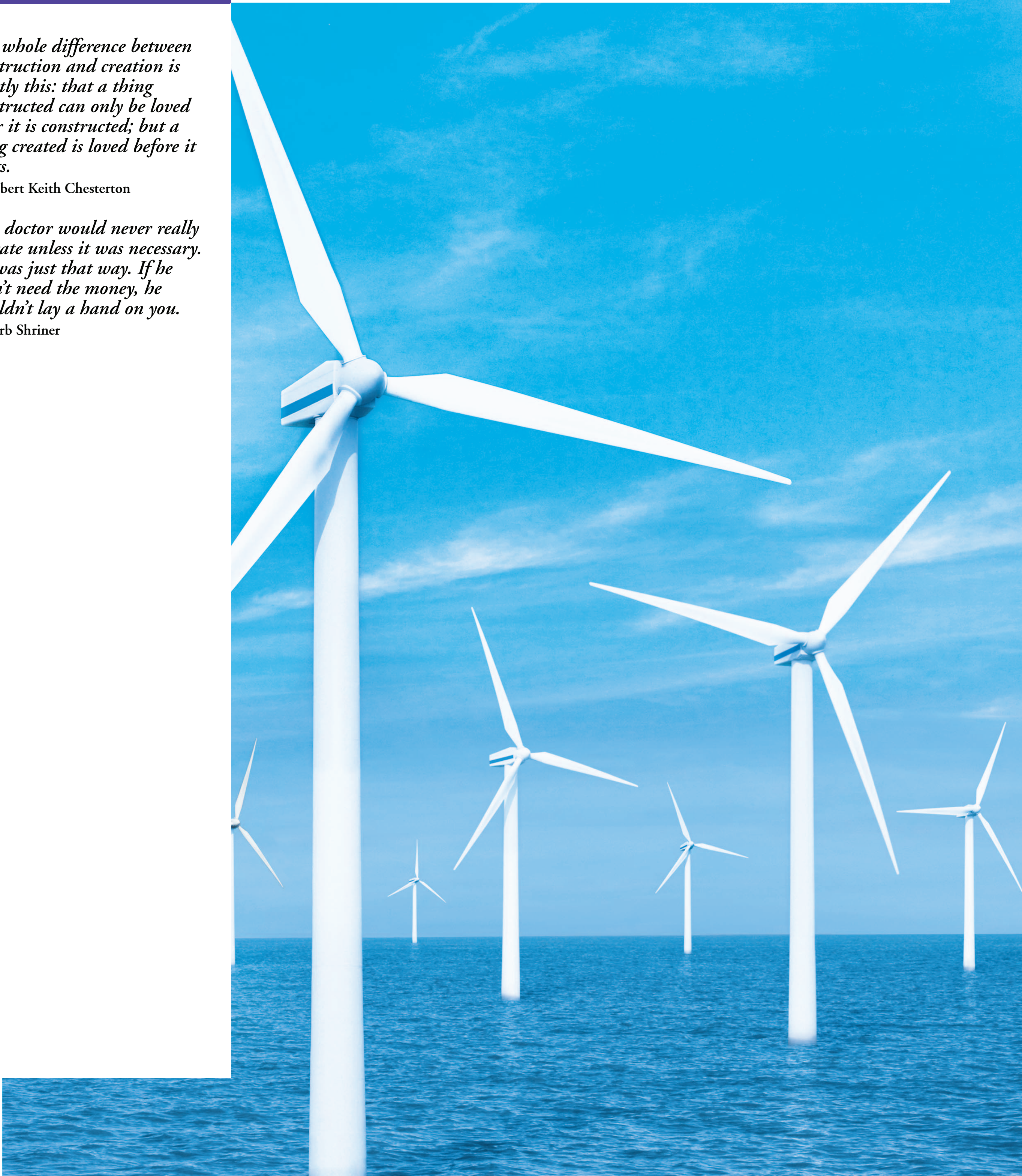
11 Operator Overloading; String and Array Objects

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

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



11.1 Case Study: String Class

As a capstone exercise to our study of overloading, we'll build our own `String` class to handle the creation and manipulation of strings (Figs. 11.1–11.3). The C++ standard library provides a similar, more robust class `string` as well. We present an example of the standard class `string` in Section 11.14 and study class `string` in detail in Chapter 18. For now, we'll make extensive use of operator overloading to craft our own class `String`.

First, we present the header file for class `String`. We discuss the private data used to represent `String` objects. Then we walk through the class's `public` interface, discussing each of the services the class provides. We discuss the member-function definitions for the class `String`. For each of the overloaded operator functions, we show the code in the program that invokes the overloaded operator function, and we provide an explanation of how the overloaded operator function works.

`String` Class Definition

Now let's walk through the `String` class header file in Fig. 11.1. We begin with the internal pointer-based representation of a `String`. Lines 55–56 declare the `private` data members of the class. Our `String` class has a `length` field, which represents the number of characters in the string, not including the null character at the end, and has a pointer `sPtr` that points to the dynamically allocated memory representing the character string.

```
1  // String.h
2  // String class definition with operator overloading.
3  #ifndef STRING_H
4  #define STRING_H
5
6  #include <iostream>
7  using namespace std;
8
9  class String
10 {
11    friend ostream &operator<<( ostream &, const String & );
12    friend istream &operator>>( istream &, String & );
13 public:
14    String( const char * = "" ); // conversion/default constructor
15    String( const String & ); // copy constructor
16    ~String(); // destructor
17
18    const String &operator=( const String & ); // assignment operator
19    const String &operator+=( const String & ); // concatenation operator
20
21    bool operator!() const; // is String empty?
22    bool operator==( const String & ) const; // test s1 == s2
23    bool operator<( const String & ) const; // test s1 < s2
24
25    // test s1 != s2
26    bool operator!=( const String &right ) const
27    {
```

Fig. 11.1 | `String` class definition with operator overloading. (Part 1 of 2.)

```
      return !( *this == right );
   } // end function operator!=

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

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

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

   char &operator[]( int ); // subscript operator (modifiable lvalue)
   char operator[]( int ) const; // subscript operator (rvalue)
   String operator()( int, int = 0 ) const; // return a substring
   int getLength() const; // return string length
private:
   int length; // string length (not counting null terminator)
   char *sPtr; // pointer to start of pointer-based string

   void setString( const char * ); // utility function
}; // end class String

#endif
```

Fig. 11.1 | `String` class definition with operator overloading. (Part 2 of 2.)

```
// String.cpp
// String class member-function and friend-function definitions.
#include <iostream>
#include <iomanip>
#include <cstring> // strcpy and strcat prototypes
#include <cstdlib> // exit prototype
#include "String.h" // String class definition
using namespace std;

// conversion (and default) constructor converts char * to String
String::String( const char *s )
   : length( ( s != 0 ) ? strlen( s ) : 0 )
{
   cout << "Conversion (and default) constructor: " << s << endl;
   setString( s ); // call utility function
} // end String conversion constructor
```

Fig. 11.2 | `String` class member-function and `friend`-function definitions. (Part 1 of 4.)

```

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

// Destructor
String::~String()
{
   cout << "Destructor: " << sPtr << endl;
   delete [] sPtr; // release pointer-based string memory
} // end ~String destructor

// overloaded = operator; avoids self assignment
const String &String::operator=( const String &right )
{
   cout << "operator= called" << endl;

   if ( &right != this ) // avoid self assignment
   {
      delete [] sPtr; // prevents memory leak
      length = right.length; // new String length
      setString( right.sPtr ); // call utility function
   } // end if
   else
      cout << "Attempted assignment of a String to itself" << endl;

   return *this; // enables cascaded assignments
} // end function operator=

// concatenate right operand to this object and store in this object
const String &String::operator+=( const String &right )
{
   size_t newLength = length + right.length; // new length
   char *tempPtr = new char[ newLength + 1 ]; // create memory

   strcpy( tempPtr, sPtr ); // copy sPtr
   strcpy( tempPtr + length, right.sPtr ); // copy right.sPtr

   delete [] sPtr; // reclaim old space
   sPtr = tempPtr; // assign new array to sPtr
   length = newLength; // assign new length to length
   return *this; // enables cascaded calls
} // end function operator+=

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

Fig. 11.2 | String class member-function and friend-function definitions. (Part 2 of 4.)

```

// 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<

// return reference to character in String as a modifiable lvalue
char &String::operator[]( int subscript )
{
   // test for subscript out of range
   if ( subscript < 0 || subscript >= length )
   {
      cerr << "Error: Subscript " << subscript
         << " out of range" << endl;
      exit( 1 ); // terminate program
   } // end if

   return sPtr[ subscript ]; // non-const return; modifiable lvalue
} // end function operator[]

// return reference to character in String as rvalue
char String::operator[]( int subscript ) const
{
   // test for subscript out of range
   if ( subscript < 0 || subscript >= length )
   {
      cerr << "Error: Subscript " << subscript
           << " out of range" << endl;
      exit( 1 ); // terminate program
   } // end if

   return sPtr[ subscript ]; // returns copy of this element
} // end function operator[]

// return a substring beginning at index and of length subLength
String String::operator()( int index, int subLength ) const
{
   // 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;

```

Fig. 11.2 | `String` class member-function and `friend`-function definitions. (Part 3 of 4.)

```
   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';

   // 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 getLength

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

   if ( string2 != 0 ) // if string2 is not null pointer, copy contents
      strcpy( sPtr, string2 ); // copy literal to object
   else // if string2 is a null pointer, make this an empty string
      sPtr[ 0 ] = '\0'; // empty string
} // end function setString

// overloaded output operator
ostream &operator<<( ostream &output, const String &s )
{
   output << s.sPtr;
   return output; // enables cascading
} // end function operator<<

// 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
} // end function operator>>
```

Fig. 11.2 | String class member-function and friend-function definitions. (Part 4 of 4.)

```cpp
// Fig. 11.11: fig11_11.cpp
// String class test program.
#include <iostream>
#include "String.h"
using namespace std;

int main()
{
   String s1( "happy" );
   String s2( " birthday" );
   String s3;

   // test overloaded equality and relational operators
   cout << "s1 is \"" << s1 << "\"; s2 is \"" << s2
      << "\"; s3 is \"" << s3 << '\"'
      << boolalpha << "\n\nThe results of comparing s2 and s1:"
      << "\ns2 == s1 yields " << ( s2 == s1 )
      << "\ns2 != s1 yields " << ( s2 != s1 )
      << "\ns2 >  s1 yields " << ( s2 > s1 )
      << "\ns2 <  s1 yields " << ( s2 < s1 )
      << "\ns2 >= s1 yields " << ( s2 >= s1 )
      << "\ns2 <= s1 yields " << ( s2 <= s1 );

   // test overloaded String empty (!) operator
   cout << "\n\nTesting !s3:" << endl;

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

   // test overloaded String concatenation operator
   cout << "\n\ns1 += s2 yields s1 = ";
   s1 += s2; // test overloaded concatenation
   cout << s1;

   // test conversion constructor
   cout << "\n\ns1 += \" to you\" yields" << endl;
   s1 += " to you"; // test conversion constructor
   cout << "s1 = " << s1 << "\n\n";

   // 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), is: "
      << s1( 15 ) << "\n\n";

```

Fig. 11.3 | String class test program. (Part 1 of 3.)

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

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

   // test destructor
   delete s4Ptr;

   // test using subscript operator to create a modifiable lvalue
   s1[ 0 ] = 'H';
   s1[ 6 ] = 'B';
   cout << "\ns1 after s1[0] = 'H' and s1[6] = 'B' is: "
      << s1 << "\n\n";

   // test subscript out of range
   cout << "Attempt to assign 'd' to s1[30] yields:" << endl;
   s1[ 30 ] = 'd'; // ERROR: subscript out of range
} // end main
```

```
Conversion (and default) constructor: happy
Conversion (and default) constructor:  birthday
Conversion (and default) constructor:
s1 is "happy"; s2 is " birthday"; s3 is ""

The results of comparing s2 and s1:
s2 == s1 yields false
s2 != s1 yields true
s2 >  s1 yields false
s2 <  s1 yields true
s2 >= s1 yields false
s2 <= s1 yields true

Testing !s3:
s3 is empty; assigning s1 to s3;
operator= called
s3 is "happy"

s1 += s2 yields s1 = happy birthday

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

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

Fig. 11.3 | String class test program. (Part 2 of 3.)

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

Destructor: to you
Copy constructor: happy birthday to you

*s4Ptr = happy birthday to you

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

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

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

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

Fig. 11.3 | `String` class test program. (Part 3 of 3.)

Overloading the Stream Insertion and Stream Extraction Operators as `friend`*s*

Lines 12–13 (Fig. 11.1) declare the overloaded stream insertion operator function `operator<<` (defined in Fig. 11.2, lines 170–174) and the overloaded stream extraction operator function `operator>>` (defined in Fig. 11.2, lines 177–183) as `friend`s of the class. The implementation of `operator<<` is straightforward. Function `operator>>` restricts the total number of characters that can be read into array `temp` to 99 with `setw` (line 180); the 100th position is reserved for the string's terminating null character. [*Note:* We did not have this restriction for `operator>>` in class `Array` (Figs. 11.6–11.7), because that class's `operator>>` read one array element at a time and stopped reading values when the end of the array was reached. Object `cin` does not know how to do this by default for input of character arrays.] Also, note the use of `operator=` (line 181) to assign the C-style string `temp` to the `String` object to which `s` refers. This statement invokes the conversion constructor to create a temporary `String` object containing the C-style string; the temporary `String` is then assigned to `s`. We could eliminate the overhead of creating the temporary `String` object here by providing another overloaded assignment operator that receives a parameter of type `const char *`.

`String` *Conversion Constructor*

Line 15 (Fig. 11.1) declares a conversion constructor. This constructor (defined in Fig. 11.2, lines 22–27) takes a `const char *` argument (that defaults to the empty string; Fig. 11.1, line 15) and initializes a `String` object containing that same character string. Any single-argument constructor can be thought of as a conversion constructor. As we'll see, such constructors are helpful when we are doing any `String` operation using `char *` arguments. The conversion constructor can convert a `char *` string into a `String` object, which can then be assigned to the target `String` object. The availability of this conversion constructor means that it isn't necessary to supply an overloaded assignment operator for specifically assigning character strings to `String` objects. The compiler invokes the conversion constructor to create a temporary `String` object containing the character string; then the overloaded assignment operator is invoked to assign the temporary `String` object to another `String` object.

Software Engineering Observation 11.1

When a conversion constructor is used to perform an implicit conversion, C++ can apply only one implicit constructor call (i.e., a single user-defined conversion) to try to match the needs of another overloaded operator. The compiler will not match an overloaded operator's needs by performing a series of implicit, user-defined conversions.

The `String` conversion constructor could be invoked in such a declaration as `String s1( "happy" )`. The conversion constructor calculates the length of its character-string argument and assigns it to data member `length` in the member-initializer list. Then, line 26 calls utility function `setString` (defined in Fig. 11.2, lines 159–167), which uses `new` to allocate a sufficient amount of memory to `private` data member `sPtr` and uses `strcpy` to copy the character string into the memory to which `sPtr` points.[1]

`String` Copy Constructor

Line 16 in Fig. 11.1 declares a copy constructor (defined in Fig. 11.2, lines 30–35) that initializes a `String` object by making a copy of an existing `String` object. As with our class `Array` (Figs. 11.6–11.7), such copying must be done carefully to avoid the pitfall in which both `String` objects point to the same dynamically allocated memory. The copy constructor operates similarly to the conversion constructor, except that it simply copies the `length` member from the source `String` object to the target `String` object. The copy constructor calls `setString` to create new space for the target object's internal character string. If it simply copied the `sPtr` in the source object to the target object's `sPtr`, then both objects would point to the same dynamically allocated memory. The first destructor to execute would then delete the dynamically allocated memory, and the other object's `sPtr` would be undefined (i.e., `sPtr` would be a dangling pointer), a situation likely to cause a serious runtime error.

`String` Destructor

Line 17 of Fig. 11.1 declares the `String` destructor (defined in Fig. 11.2, lines 38–42). The destructor uses `delete []` to release the dynamic memory to which `sPtr` points.

1. There is a subtle issue in the implementation of this conversion constructor. As implemented, if a null pointer (i.e., `0`) is passed to the constructor, the program will fail. The proper way to implement this constructor would be to detect whether the constructor argument is a null pointer, then "throw an exception." Chapter 16 discusses how we can make classes more robust in this manner. Also, a null pointer (`0`) is not the same as the empty string (`""`). A null pointer is a pointer that does not point to anything. An empty string is an actual string that contains only a null character (`'\0'`).

Overloaded Assignment Operator

Line 19 (Fig. 11.1) declares the overloaded assignment operator function `operator=` (defined in Fig. 11.2, lines 45–59). When the compiler sees an expression like `string1 = string2`, the compiler generates the function call

```
      string1.operator=( string2 );
```

The overloaded assignment operator function `operator=` tests for self-assignment. If this is a self-assignment, the function does not need to change the object. If this test were omitted, the function would immediately delete the space in the target object and thus lose the character string, such that the pointer would no longer be pointing to valid data—a classic example of a dangling pointer. If there is no self-assignment, the function deletes the memory and copies the `length` field of the source object to the target object. Then `operator=` calls `setString` to create new space for the target object and copy the character string from the source object to the target object. Whether or not this is a self-assignment, `operator=` returns `*this` to enable cascaded assignments.

Overloaded Addition Assignment Operator

Line 20 of Fig. 11.1 declares the overloaded string-concatenation operator `+=` (defined in Fig. 11.2, lines 62–74). When the compiler sees the expression `s1 += s2` (line 40 of Fig. 11.3), the compiler generates the member-function call

```
      s1.operator+=( s2 )
```

Function `operator+=` calculates the combined length of the concatenated string and stores it in local variable `newLength`, then creates a temporary pointer (`tempPtr`) and allocates a new character array in which the concatenated string will be stored. Next, `operator+=` uses `strcpy` to copy the original character strings from `sPtr` and `right.sPtr` into the memory to which `tempPtr` points. The location into which `strcpy` will copy the first character of `right.sPtr` is determined by the pointer-arithmetic calculation `tempPtr + length`. This calculation indicates that the first character of `right.sPtr` should be placed at location `length` in the array to which `tempPtr` points. Next, `operator+=` uses `delete []` to release the space occupied by this object's original character string, assigns `tempPtr` to `sPtr` so that this `String` object points to the new character string, assigns `newLength` to `length` so that this `String` object contains the new string length and returns `*this` as a `const String &` to enable cascading of `+=` operators.

Do we need a second overloaded concatenation operator to allow concatenation of a `String` and a `char *`? No. The `const char *` conversion constructor converts a C-style string into a temporary `String` object, which then matches the existing overloaded concatenation operator. This is exactly what the compiler does when it encounters line 45 in Fig. 11.3. Again, C++ can perform such conversions only one level deep to facilitate a match. C++ can also perform an implicit compiler-defined conversion between fundamental types before it performs the conversion between a fundamental type and a class. When a temporary `String` object is created in this case, the conversion constructor and the destructor are called (see the output resulting from line 45, `s1 += " to you"`, in Fig. 11.3). This is an example of function-call overhead that is hidden from the client of the class when temporary class objects are created and destroyed during implicit conversions. Similar overhead is generated by copy constructors in call-by-value parameter passing and in returning class objects by value.

Performance Tip 11.1

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

Software Engineering Observation 11.2

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

Overloaded Negation Operator

Line 22 of Fig. 11.1 declares the overloaded negation operator (defined in Fig. 11.2, lines 77–80). This operator determines whether an object of our `String` class is empty. For example, when the compiler sees the expression `!string1`, it generates the function call

```
string1.operator!()
```

This function simply returns the result of testing whether `length` is equal to zero.

Overloaded Equality and Relational Operators

Lines 23–24 of Fig. 11.1 declare the overloaded equality operator (defined in Fig. 11.2, lines 83–86) and the overloaded less-than operator (defined in Fig. 11.2, lines 89–92) for class `String`. These are similar, so let's discuss only one example, namely, overloading the == operator. When the compiler sees the expression `string1 == string2`, the compiler generates the member-function call

```
string1.operator==( string2 )
```

which returns `true` if `string1` is equal to `string2`. Each of these operators uses function `strcmp` (from `<cstring>`) to compare the character strings in the `String` objects. Many C++ programmers advocate using some of the overloaded operator functions to implement others. So, the `!=`, `>`, `<=` and `>=` operators are implemented (Fig. 11.1, lines 27–48) in terms of `operator==` and `operator<`. For example, overloaded function `operator>=` (implemented in lines 45–48 in the header file) uses the overloaded `<` operator to determine whether one `String` object is greater than or equal to another. The operator functions for `!=`, `>`, `<=` and `>=` are defined in the header file. The compiler inlines these definitions to eliminate the overhead of the extra function calls.

Software Engineering Observation 11.3

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

Overloaded Subscript Operators

Lines 50–51 in the header file declare two overloaded subscript operators (defined in Fig. 11.2, lines 95–106 and 109–120, respectively)—one for non-`const` `String`s and one for const Strings. When the compiler sees an expression like string1[0], the compiler generates the member-function call

```
      string1.operator[]( 0 )
```

(using the appropriate version of operator[] based on whether the String is const). Each implementation of operator[] first validates the subscript to ensure that it's in range. If the subscript is out of range, each function prints an error message and terminates the program with a call to exit.[2] If the subscript is in range, the non-const version of operator[] returns a char & to the appropriate character of the String object; this char & may be used as an *lvalue* to modify the designated character of the String object. The const version of operator[] returns the appropriate character of the String object; this can be used only as an *rvalue* to read the value of the character.

Error-Prevention Tip 11.1

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

Overloaded Function Call Operator

Line 52 of Fig. 11.1 declares the **overloaded function call operator** (defined in Fig. 11.2, lines 123–150). We overload this operator to select a substring from a String. The two integer parameters specify the start location and the length of the substring being selected from the String. If the start location is out of range or the substring length is negative, the operator simply returns an empty String. If the substring length is 0, then the substring is selected to the end of the String object. For example, suppose string1 is a String object containing the string "AEIOU". For the expression string1(2, 2), the compiler generates the member-function call

```
      string1.operator()( 2, 2 )
```

When this call executes, it produces a String object containing the string "IO" and returns a copy of that object.

Overloading the function call operator () is powerful, because functions can take arbitrarily long and complex parameter lists. So we can use this capability for many interesting purposes. One such use of the function call operator is an alternate array-subscripting notation: Instead of using C's awkward double-square-bracket notation for pointer-based two-dimensional arrays, such as in a[b][c], some programmers prefer to overload the function call operator to enable the notation a(b, c). The overloaded function call operator must be a non-static member function. This operator is used only when the "function name" is an object of class String.

String Member Function **getLength**

Line 53 in Fig. 11.1 declares function getLength (defined in Fig. 11.2, lines 153–156), which returns the length of a String.

2. Again, it's more appropriate when a subscript is out of range to "throw an exception" indicating the out-of-range subscript.

*Notes on Our **`String`** Class*

At this point, you should step through the code in `main`, examine the output window and check each use of an overloaded operator. As you study the output, pay special attention to the implicit constructor calls that are generated to create temporary `String` objects throughout the program. Many of these calls introduce additional overhead into the program that can be avoided if the class provides overloaded operators that take `char *` arguments. However, additional operator functions can make the class harder to maintain, modify and debug.