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

Fig. 10.5 | Overloaded stream insertion and stream extraction operators. (Part 2 of 2.)

10.5 Overloading the Binary Stream Insertion and Stream Extraction Operators (cont.)

Overloading the Stream Extraction (>>) Operator

- The stream extraction operator function `operator>>` (Fig. 10.4, lines 21–30) takes `istream` reference `input` and `PhoneNumber` reference `number` as arguments and returns an `istream` reference.
- Operator function `operator>>` inputs phone numbers of the form
  - `(800) 555-1212`
- When the compiler sees the expression
  - `cin >> phone`
- In line 16 of Fig. 10.5, the compiler generates the *non-member function call*
  - `operator>>( cin, phone );`
- When this call executes, reference parameter `input` (Fig. 10.4, line 21) becomes an alias for `cin` and reference parameter `number` becomes an alias for `phone`.

10.5 Overloading the Binary Stream Insertion and Stream Extraction Operators (cont.)

- The operator function reads as `string`s the three parts of the telephone number into the `areaCode` (line 24), `exchange` (Line 26) and `line` (line 28) members of the `PhoneNumber` object referenced by parameter `Number`.
- Stream manipulator `setw` limits the number of characters read into each `string`.
- The parentheses, space and dash characters are skipped by calling `istream` member function `ignore` (Fig. 10.4, lines 23, 25 and 27), which discards the specified number of characters in the input stream (one character by default).

10.5 Overloading Stream Insertion and Stream Extraction Operators (cont.)

- Function `operator>>` returns `istream` reference `input` (i.e., `cin`).
- This enables input operations on `PhoneNumber` objects to be cascaded with input operations on other `PhoneNumber` objects or on objects of other data types.

Good Programming Practice 10.1

Overloaded operators should mimic the functionality of their built-in counterparts—e.g., the + operator should perform addition, not subtraction. Avoid excessive or inconsistent use of operator overloading, as this can make a program cryptic and difficult to read.

10.5 Overloading Stream Insertion and Stream Extraction Operators (cont.)

Overloading the Stream Insertion (<<) Operator

- The stream insertion operator function (Fig. 10.4, lines 11-16) takes an `ostream` reference (`output`) and a `const PhoneNumber` reference (`number`) as arguments and returns an `ostream` reference.
- Function `operator<<` displays objects of type `PhoneNumber`.
- When the compiler sees the expression
  - `cout << phone`

  in line 22 if Fig. 10.5, the compiler generates the non-member function call
  - `operator<<( cout, phone );`
- Function `operator<<` displays the parts of the telephone number as `string`s, because they're stored as `string` objects.

10.5 Overloading the Binary Stream Insertion and Stream Extraction Operators (cont.)

Overloaded Operators as Non-Member `friend` Functions

- The functions `operator>>` and `operator<<` are declared in `PhoneNumber` as non-member, `friend` functions.
- They're *non-member functions* because the object of class `PhoneNumber` is the operator's *right* operand.

Software Engineering Observation 10.2

New input/output capabilities for user-defined types are added to C++ without modifying standard input/output library classes. This is another example of C++'s extensibility.

10.5 Overloading the Binary Stream Insertion and Stream Extraction Operators (cont.)

Why Overloaded Stream Insertion and Stream Extraction Operators Are Overloaded as Non-Member Functions

- The overloaded stream insertion operator (<<) is used in an expression in which the left operand has type `ostream &`, as in `cout << classObject`.
- To use the operator in this manner where the *right* operand is an object of a user-defined class, it must be overloaded as a *non-member function*.

10.5 Overloading the Binary Stream Insertion and Stream Extraction Operators (cont.)

- Similarly, the overloaded stream extraction operator (`>>`) is used in an expression in which the left operand has type `istream &`, as in `cin >> classObject`, and the *right* operand is an object of a user-defined class, so it, too, must be a non-member function.
- Each of these overloaded operator functions may require access to the `private` data members of the class object being output or input, so these overloaded operator functions can be made `friend` functions of the class for performance reasons.

10.6 Overloading Unary Operators

- *A unary operator for a class can be overloaded as a non-`static` member function with no arguments or as a non-member function with one argument that must be an object (or a reference to an object) of the class.*
- A unary operator such as `!` may be overloaded as a *non-member function* with one parameter.

10.7 Overloading the Unary Prefix and Postfix ++ and −− Operators

- The prefix and postfix versions of the increment and decrement operators can all be overloaded.
- *To overload the increment operator to allow both prefix and postfix increment usage, each overloaded operator function must have a distinct signature, so that the compiler will be able to determine which version of ++ is intended.*
- The prefix versions are overloaded exactly as

10.7 Overloading the Unary Prefix and Postfix ++ and -- Operators (cont.)

- Suppose that we want to add 1 to the day in `Date` object `d1`.
- When the compiler sees the preincrementing expression `++d1`, the compiler generates the *member-function call*
  - `d1.operator++()`
- The prototype for this operator function would be
  - `Date &operator++();`
- If the prefix increment operator is implemented as a non-member function, then, when the compiler sees the expression `++d1`, the compiler generates the function call
  - `operator++( d1 )`
- The prototype for this operator function would be declared in the `Date` class as
  - `Date &operator++( Date & );`

10.7 Overloading the Unary Prefix and Postfix ++ and -- Operators (cont.)

Overloading the Postfix Increment Operator

- Overloading the postfix increment operator presents a challenge, because the compiler must be able to distinguish between the signatures of the overloaded prefix and postfix increment operator functions.
- The *convention* that has been adopted in C++ is that, when the compiler sees the postincrementing expression `d1++`, it generates the *member-function call*
  - `d1.operator++( 0 )`
- The prototype for this function is
  - `Date operator++( int )`
- The argument `0` is strictly a "dummy value" that enables the compiler to distinguish between the prefix and postfix increment operator functions.
- The same syntax is used to differentiate between the prefix and postfix decrement operator functions.

10.7 Overloading the Unary Prefix and Postfix ++ and -- Operators (cont.)

- If the postfix increment is implemented as a non-member function, then, when the compiler sees the expression `d1++`, the compiler generates the function call
  - `operator++( d1, 0 )`
- The prototype for this function would be
  - `Date operator++( Date &, int );`
- Once again, the `0` argument is used by the compiler to distinguish between the prefix and postfix increment operators implemented as non-member functions.
- The postfix increment operator returns `Date` objects *by value*, whereas the prefix increment operator returns `Date` objects *by reference*—the postfix increment operator typically returns a temporary object that contains the original value of the object before the increment occurred.