10.3 Fundamentals of Operator Overloading

- As you saw in Fig. 10.1, operators provide a concise notation for manipulating string objects.
- You can use operators with your own user-defined types as well.
- Although C++ does not allow new operators to be created, it does allow most existing operators to be overloaded so that, when they're used with objects, they have meaning appropriate to those objects.

10.3 Fundamentals of Operator Overloading (cont.)

- Operator overloading is not automatic—you must write operator-overloading functions to perform the desired operations.
- An operator is overloaded by writing a non-`static` member function definition or non-member function definition as you normally would, except that the function name starts with the keyword `operator` followed by the symbol for the operator being overloaded.
  - For example, the function name `operator+` would be used to overload the addition operator (+) for use with objects of a particular class (or

10.3 Fundamentals of Operator Overloading (cont.)

- When operators are overloaded as member functions, they must be non-`static`, because *they must be called on an object of the class* and operate on that object.
- To use an operator on class objects, you must define overloaded operator functions for that class—with three exceptions.
  - The *assignment operator (=)* may be used with *most* classes to perform *memberwise assignment* of the data members—each data member is assigned from the assignment's "source" object (on the right) to the "target" object (on the left).
    - *Memberwise assignment is dangerous for classes with pointer members*, so we'll explicitly overload the assignment operator for such classes.
  - The *address operator (&)* returns a pointer to the object; this operator also can be overloaded.
  - The *comma operator* evaluates the expression to its left then the expression to its right, and returns the value of the latter expression.

10.3 Fundamentals of Operator Overloading (cont.)

- Most of C++’s operators can be overloaded.
- Figure 10.2 shows the operators that cannot be overloaded.

Operators that cannot be overloaded			
.	.* (pointer to member)	::	?:

Fig. 10.2 | Operators that cannot be overloaded.

10.3 Fundamentals of Operator Overloading (cont.)

- The *precedence of an operator cannot be changed by overloading*.
  - However, parentheses can be used to force the order of evaluation of overloaded operators in an expression.
- *The associativity of an operator cannot be changed by overloading*
  - if an operator normally associates from left to right, then so do all of its overloaded versions.
- *You cannot change the "arity" of an operator* (that is, the number of operands an operator takes)
  - overloaded unary operators remain unary operators; overloaded binary operators remain binary operators. Operators &, *, + and - all have both unary and binary versions; these unary and binary versions can be separately overloaded.

10.3 Fundamentals of Operator Overloading (cont.)

- *You cannot create new operators; only existing operators can be overloaded.*
- The meaning of how an operator works on values of fundamental types *cannot* be changed by operator overloading.
  - For example, you cannot make the + operator subtract two `int`s. Operator overloading works only with *objects of user-defined types or with a mixture of an object of a user-defined type and an object of a fundamental type.*

10.3 Fundamentals of Operator Overloading (cont.)

- Related operators, like + and +=, must be overloaded separately.
- When overloading `()`, `[]`, `->` or any of the assignment operators, the operator overloading function must be declared as a class member.
  - For all other overloadable operators, the operator overloading functions can be member functions or non-member functions.

Software Engineering Observation 10.1

Overload operators for class types so they work as closely as possible to the way built-in operators work on fundamental types.

10.4 Overloading Binary Operators

- *A binary operator can be overloaded as a non-`static` member function with one parameter or as a non-member function with two parameters (one of those parameters must be either a class object or a reference to a class object).*
- As a non-member function, binary operator `<` must take two arguments—one of which must be an object (or a reference to an object) of the class.

10.5 Overloading the Binary Stream Insertion and Stream Extraction Operators

- You can input and output fundamental-type data using the stream extraction operator `>>` and the stream insertion operator `<<`.
- The C++ class libraries overload these binary operators for each fundamental type, including pointers and `char *` strings.
- You can also overload these operators to perform input and output for your own types.
- The program of Figs. 10.3–10.5 overloads these operators to input and output `PhoneNumber` objects in the format "`(000) 000-0000`." The program assumes telephone numbers are input correctly.

```cpp
// Fig. 10.3: PhoneNumber.h
// PhoneNumber class definition
#ifndef PHONENUMBER_H
#define PHONENUMBER_H

#include <iostream>
#include <string>

class PhoneNumber
{
   friend std::ostream &operator<<( std::ostream &, const PhoneNumber & );
   friend std::istream &operator>>( std::istream &, PhoneNumber & );
private:
   std::string areaCode; // 3-digit area code
   std::string exchange; // 3-digit exchange
   std::string line; // 4-digit line
}; // end class PhoneNumber

#endif
```

Fig. 10.3 | PhoneNumber class with overloaded stream insertion and stream extraction operators as `friend` functions.

```cpp
// Fig. 10.4: PhoneNumber.cpp
// Overloaded stream insertion and stream extraction operators
// for class PhoneNumber.
#include <iomanip>
#include "PhoneNumber.h"
using namespace std;

// overloaded stream insertion operator; cannot be
// a member function if we would like to invoke it with
// cout << somePhoneNumber;
ostream &operator<<( ostream &output, const PhoneNumber &number )
{
   output << "(" << number.areaCode << ") "
      << number.exchange << "-" << number.line;
   return output; // enables cout << a << b << c;
} // end function operator<<

```

Fig. 10.4 | Overloaded stream insertion and stream extraction operators for class PhoneNumber. (Part 1 of 2.)

```cpp
// overloaded stream extraction operator; cannot be
// a member function if we would like to invoke it with
// cin >> somePhoneNumber;
istream &operator>>( istream &input, PhoneNumber &number )
{
   input.ignore(); // skip (
   input >> setw( 3 ) >> number.areaCode; // input area code
   input.ignore( 2 ); // skip ) and space
   input >> setw( 3 ) >> number.exchange; // input exchange
   input.ignore(); // skip dash (-)
   input >> setw( 4 ) >> number.line; // input line
   return input; // enables cin >> a >> b >> c;
} // end function operator>>
```

Fig. 10.4 | Overloaded stream insertion and stream extraction operators for class PhoneNumber. (Part 2 of 2.)

```cpp
// Fig. 10.5: fig10_05.cpp
// Demonstrating class PhoneNumber's overloaded stream insertion
// and stream extraction operators.
#include <iostream>
#include "PhoneNumber.h"
using namespace std;

int main()
{
   PhoneNumber phone; // create object phone

   cout << "Enter phone number in the form (123) 456-7890:" << endl;

   // cin >> phone invokes operator>> by implicitly issuing
   // the non-member function call operator>>( cin, phone )
   cin >> phone;

   cout << "The phone number entered was: ";

   // cout << phone invokes operator<< by implicitly issuing
   // the non-member function call operator<<( cout, phone )
   cout << phone << endl;
} // end main
```

Fig. 10.5 | Overloaded stream insertion and stream extraction operators. (Part 1 of 2.)