More Class Features & Other Types

11

REVIEW QUESTIONS

1. An in-line function can be called recursively.

b. false

3. Overloading is the definition of a two or more classes with the same name.

b. false

5. An integer value can be assigned to only one enumeration constant in an enumerated type.

b. false

7. The selection operator is used with a pointer to access individual fields in a structure.

a. true

9. The _____ can be used to create a new type that can be used anywhere a type is permitted.

d. type definition

11. Which of the following statements about enumerated types is true?

d. Enumerated types are automatically assigned constant values unless otherwise directed.

13. A(n) _____ is a construct that allows a portion of memory to be shared by different types of data.

b. union

EXERCISES

15. The conditional operator (?:) cannot be overloaded.

17. First, this must be a friend function. Second, while this would work, it is a poor design. Binary operators should return a value. This one returns void. From its design, we conclude that it is adding x to fun, which would be better designed using the += operator.

19. This definition violates the binary arity operator (brackets) by requiring three operands, the member class (Fun), a second class instance, fun, and the integer x.

21. There are two errors. First, the bracket is a binary operator. The definition defines three operands, the invoking class and two parameters in the calling sequence. Second, the function header is wrong.

23. The member operator (.) cannot be overloaded.

25. The difference between these two functions is that the first one takes as its parameter an object of the Fun class while the second one takes an int. In the first case, the value in the parameter object (fun) is added to and placed in the member object (z) as shown below.

```
fun1+= fun2;
```

In the second case, an integer value is added to the member object as shown below.

```
fun1+= x;
```

27. The equivalent expressions are:

a. y && z

b. y == z

c. y += x

d. ++z;

e. x[6]

f. –x

g. y < x

h. x (4)

i. y + z

29. A structure for one array element and the array definition are shown below.

```
struct MONTH
   {
    short  month;
    float  dailySales[31];
    short  days;
   };
MONTH month_array[12] = {
                          { 1, {0.0}, 31 },
                          { 2, {0.0}, 28 },
                           ...
                          { 12, {0.0}, 31 }
                         };  // MONTH definition
cout << month_array[i].month << endl
     << month_array[i].dailySales[j] << endl
     << month_array[i].days;
```

31. The output is:

12.45

B

23.34

PROBLEMS

33.

```
/* ============= Fraction :: operator *= ============
   Multiply two fractions
      Pre  fractions contain values
      Post product stored in calling fraction
*/
void Fraction :: operator*= (const Fraction& fr2)
{
   numerator    *= fr2.numerator;
   denominator  *= fr2.denominator;
   *this         = Fraction (numerator, denominator);
   return;
}  // Fraction operator*=
```

35.

```
/* ================== operator - ==================
   Subtract one Fraction from another
      Pre  fractions contain values
      Post difference returned
*/
Fraction operator- (const Fraction& fr1,
                    const Fraction& fr2)
{
   int numen =
             (fr1.numerator   * fr2.denominator)
           - (fr2.numerator   * fr1.denominator);

   int denom = fr1.denominator * fr2.denominator;
   return Fraction(numen, denom);
}  // Fraction friend operator-
```

37.

```
/* ================== operator / ================
   Divide one fraction by another
      Pre  fractions contain values
      Post quotient returned
*/
Fraction operator/ (const Fraction& fr1,
                    const Fraction& fr2)
{
   int numen = fr1.numerator   * fr2.denominator;
   int denom = fr1.denominator * fr2.numerator;
   return Fraction(numen, denom);
}  // Fraction friend operator/
```

39.

```
/* ================== operator < ==============
   Determine if 1 fraction is less than another
      Pre  fractions contain values
      Post returns true  if fr1 < fr2
           returns false if fr1 >= fr2
*/
bool operator< (const Fraction& fr1,
                const Fraction& fr2)
{
   if   (fr1.numerator   * fr2.denominator
      <  fr1.denominator * fr2.numerator)
         return true;
```

```
   return false;
}  // Fraction friend operator<
```

41.

```
/* ======== Fraction :: operator += (integer) ========
   Add an integer to a fraction
      Pre  fraction contains value
      Post sum stored in calling fraction
*/
void Fraction :: operator+= (const int number)
{
   numerator += (number * denominator);
   *this      =  Fraction (numerator, denominator);
   return;
}  // Fraction operator+= (integer)
```

43.

```
/* ================= operator && =================
   Determine if either fraction contains non-0 value
      Pre  fractions contain values
      Post returns true  if fr1 and fr2 are not zero
           returns false if fr1 or fr2 is zero
*/
bool operator&& (const Fraction& fr1,
                 const Fraction& fr2)
{
   if ((fr1.numerator != 0) && (fr2.numerator != 0))
      return true;
   return false;
}  // Fraction friend operator &&
```

45.

```
/* ================ operator () =================
   Extracts the integral part of a fraction
      Pre  fraction contains value
           n is dummy argument to satisfy binary arity
      Post returns integral part of a fraction
*/
int Fraction :: operator() (int n)
{
   int result;
   result  =  numerator / denominator;
   return result;
}  // Fraction friend operator ()
```

47.

```
class Fraction
   {
    …
   public :
      Fraction ()
      {
         numerator   = 0;
         denominator = 1;
      } // default constructor
    …
   };  // class Fraction
```

49.

```
class Fraction
{
   …
   public :
      ~Fraction () { }
   …
}; // class Fraction
```

51.

```
class Bills
{
   private:
      struct Denominations
        {
         int d100s;
         int d50s;
         int d20s;
         int d10s;
         int d5s;
         int d1s;
        }; // Denominations
      Denominations denoms;

   public:
           Bills (int total = 0);
      void print ();
};  // class Bills

// ================ Bills constructor ==============
Bills :: Bills (int total)
{
   if (total)
      {
       int rem      = total;
       denoms.d100s =  rem / 100;
       rem         %= 100;
       denoms.d50s  =  rem / 50;
       rem         %= 50;
       denoms.d20s  =  rem / 20;
       rem         %= 20;
       denoms.d10s  =  rem / 10;
       rem         %= 10;
       denoms.d5s   =  rem / 5;
       rem         %= 5;
       denoms.d1s   =  rem / 1;
       rem         %= 1;
      } // if
   else
      denoms.d100s
        = denoms.d50s
        = denoms.d20s
        = denoms.d10s
        = denoms.d5s
        = denoms.d1s
        = 0;
   return;
}  // Bills constructor
```

```
void Bills :: print ()
{
   cout << "100s: "     << setw(5)
        << denoms.d100s << endl;
   cout << " 50s: "     << setw(5)
        << denoms.d50s  << endl;
   cout << " 20s: "     << setw(5)
        << denoms.d20s  << endl;
   cout << " 10s: "     << setw(5)
        << denoms.d10s  << endl;
   cout << "  5s: "     << setw(5)
        << denoms.d5s   << endl;
   cout << "  1s: "     << setw(5)
        << denoms.d1s   << endl;
   return;
}  // print Bills
```