Cairo University
Faculty of Computers and Information
Computer Science Department

Programming-2 CS213
2018/2019

Introduction

Dr. Amin Allam

1 Structured programming

Consider the following definitions of `Fraction` data type and associated functions:

```
struct Fraction
{
   int num;    // numerator
   int den;    // denominator
};

void Initialize(Fraction* f, int n, int d=1)
{
   if(d==0) d=1; // Avoid division by zero
   f->num = n;
   f->den = d;
}

double ConvertToDecimal(Fraction* f)
{
   double v = (double) f->num / f->den;
   return v;
}

Fraction Add(Fraction* a, Fraction* b)  // Add fraction a to fraction b
{                                       //  and return the result
   int new_den = a->den * b->den;
   int new_num = a->num * b->den + b->num * a->den;
   Fraction c;
   Initialize(&c, new_num, new_den);
   return c;
}

void Output(Fraction* f)  // Print fraction f to screen
{
   cout << f->num << "/" << f->den;
}
```

Consider the following program which uses the previously defined data type and functions:

```
int main()
{
   Fraction a, b, c, d;

   Initialize(&a, 0);    // Initialize fraction a to zero
   Initialize(&b, 2, 3);    // Initialize fraction b to 2/3
   Initialize(&c, 7, 4);    // Initialize fraction b to 7/4
   Initialize(&d, 0);    // Initialize fraction d to zero

   Output(&a); cout<<endl; // Prints 0/1
   Output(&b); cout<<endl; // Prints 2/3
   Output(&c); cout<<endl; // Prints 7/4

   d=Add(&b, &c);
   Output(&d); cout<<endl; // Prints 29/12
   cout << ConvertToDecimal(&d) << endl; // Prints 2.42

   return 0;
}
```

As shown in the previous program, the `Initialize()` function should be called for each constructed `Fraction` object. The benefit of this initialization is to make sure that the fraction denominator value is never zero, so as to keep each object into a valid state all the time, which avoids unexpected errors such as division by zero error in the functions `ConvertToDecimal()` and `Add()`.

The following program is syntactically valid, and will compile successfully. However, it may cause the run-time division by zero error while execution:

```
int main()
{
   Fraction a;
   cout << ConvertToDecimal(&a) << endl; ✖ // Compiles, but may cause
                                           // runtime division by zero error
   Fraction b;
   Initialize(&b, 2, 3);
   b.den = 0; ✖ // Compiles, but moves fraction b into an invalid state
   cout << ConvertToDecimal(&b) << endl; ✖ // Compiles, but causes
                                           // runtime division by zero error
   return 0;
}
```

The following section illustrates an alternative C++ syntax which achieves the same functionality described in this section while avoiding all mentioned problems.

2 Object oriented programming

The following C++ program achieves the same functionality described in the previous section while avoiding all mentioned problems:

```
class Fraction
{
private:
   int num;    // numerator;
   int den;    // denominator;

public:

   Fraction(int n, int d=1)  // The constructor is called whenever
   {                           //  an object is constructed
      if(d==0) d=1;  // Avoid division by zero
      this->num = n;
      this->den = d;
   }

   double ConvertToDecimal()
   {
      double v = (double) this->num / this->den;
      return v;
   }

   Fraction Add(Fraction* b)   // Add the fraction object which called Add() to b
   {
      int new_den = this->den * b->den;
      int new_num = this->num * b->den + b->num * this->den;

      Fraction c(new_num, new_den);
      return c;
   }

   void Output()
   {
      cout << this->num << "/" << this->den;
   }
};
```

The above code is almost equivalent to the code of the previous section. Main differences are: (1) Data and functions are grouped into one syntactic unit: `class`. (2) The `private` and `public` modifiers. (3) The first `Fraction*` parameter of each function is removed, and accessed by the keyword `this`. (4) The `Initialize()` function is replaced by the constructor `Fraction()`.

Consider the following program which uses the previously defined class:

```
int main()
{
   Fraction a(0), b(2,3), c(7,4), d(0);   // Objects are initialized by
                                          // the constructor Fraction()
   a.Output(); cout<<endl; // Prints 0/1
   b.Output(); cout<<endl; // Prints 2/3
   c.Output(); cout<<endl; // Prints 7/4

   d=b.Add(&c); // Add the fraction object b to the fraction object c
            // b is accessed through 'this' keyword, and c is passed as parameter
   d.Output(); cout<<endl; // Prints 29/12
   cout << d.ConvertToDecimal() << endl; // Prints 2.42

   return 0;
}
```

A variable of type `Fraction` is called a `Fraction` object, or instance of `Fraction`. All `Fraction` objects are initialized by the constructor. The programmer does not need to call an `Initialize()` function as before, but he must provide valid parameter values for the constructor when the object is created. All `Fraction` class member functions must be called through a `Fraction` object followed by the dot operator followed by the function name. Alternatively, class member functions can be accessed through a `Fraction` pointer followed by the arrow operator followed by the function name. A pointer to the object preceding the dot operator can be accessed in the called function body through the keyword `this`. Consider the following codes:

```
Fraction a; ✖ // Does not compile because no enough constructor parameters
```

```
Fraction b(2,3);
b.den = 0; ✖ // Does not compile because den is private data member
```

Each `Fraction` object must be initialized by the constructor (because the constructor is called whenever an object is created). Also, all `private` data members can not be accessed from any function which is not member of the class `Fraction`. Therefore, we are sure that at any point in the life-time of any `Fraction` object, the value of `den` is never zero, and hence a division by zero error can never occur.

A division by zero error may occur only if one of the `Fraction` member functions mistakenly sets it to zero, which is a mistake by the programmer who implemented class `Fraction`, not the programmer who uses it. Therefore, if class `Fraction` is implemented correctly, it can never cause a division by zero error regardless of what the users of this class do.

Grouping data and functions in the same syntactic unit is called encapsulation, which improves readability. Controlling access to class elements from outside the class (by `private`) is called information hiding, which improves safety and reliability by pushing responsibilities from client code (programmers who use the class) down to server code (programmers who create the class).