School of Computing and Informatics

BCS362 – Generic Programming with C++

Chapter 2-1 – Composition

1 Introduction

Composition is when an object of another class is used as a member in another class. For instance, in Listing 2.

```
class Name
{
private:
  string firstName = "uRaphie";
  string lastName = "Test";
  string salutation = "Mr";
public:
  Name(){}
  Name(string fn)
  {
    firstName = fn;
  }
  void setLastName(string ln){ lastName = ln; }
  string getLastName(){ return lastName; }
  string toString()
  {
    ostringstream out;
    out << salutation << ". " << lastName << " " << firstName << endl;
    return out.str();
  }
};
class Date
{
private:
  int day;
  int month;
  int year;
public:
  Date(){}
  Date(int d, int m, int y) : day{ d }, month{ m }, year{ y }{}
  void setDay(int d){ day = d; }
  int getDay(){ return day; }
  string toString()
  {
    ostringstream out;
    out << day << "/" << month << "/" << year << endl;
    return out.str();
  }
```

```
};
```

Listing 1: Name as composition of strings and Date class

Name is an entity that consist of objects of template class string. When a class members are objects of another class, we say that class is made up of composition of other classes. Composition models a *has-a* relationship. Using class Name and class Date above, we can define a class Person using composition as

```
class Person
{
private:
  Name name; //name is an object of class Name
  Date dob; //dob is an object of class Date
public:
  Person(){}
  Person(Name n) : name{ n }{}
  Person(Name n, Date d) :name{ n }, dob{ d }{}
  void setName(Name n){ name = n; }
  Name getName(){ return name; }
  void setDoB(Date d){ dob = d; }
  Date getDoB(){ return dob; }
  string toString()
  {
    ostringstream out;
    out << "Name : " << name.toString() << "Date of birth : " << dob.toString();
    return out.str();
  }
};
```

Listing 2: Person as composition of Name and Date

Listing 3 show the main function to test our classes

```
int main()
{
  Person p;
  Name name;
  Date date(23, 3, 1995);
  name.setLastName("uDoktela");
  p.setName(name);
  p.setDoB(date);
  cout << "Person Details are: \n" << p.toString();
  cin.get();
  return 0;
}
```

Listing 3: Person as composition of Name and Date

and the output of this code is

```
Person Details are:
Name : Mr. uDoktela uRaphie
Date of birth : 23/3/1995
```

Figure 1: Output of code in Listing 3

2 Using the this pointer

There is only one copy of class functionality, but there can be several objects of a class. So how does member functions know which object's data members to manipulate? Every object has access to its own address

through a pointer called this.

The this pointer is not part of the object itself - the memory occupied by the this pointer is not reflected in the result of sizeof operation on a pointer. The this pointer is implicitly passed by the compiler to each of the objects non-static member functions. The this pointer can be used in a number of ways.

2.1 Using this pointer to avoid name collision

The this pointer can be used to avoid name collision when a member function or constructor parameter has the same name as a member variable. The code in Listing 2 (Date class) could have a function to set year defined as

```
1 void setMonth(int month)
2 {
3   if (month > 1 && month <= 12)
4     this->month = month;
5   else
6     throw invalid_argument("Month must be between 1 -- 12");
7 }
```

Listing 4: Using this to avoid name collision

The this pointer on line 4 tells the compiler the month on left of = is referring to a member of this class.

2.2 Using this pointer to enable cascaded function calls

This pointer can also be used to enable cascaded function calls (invoking several functions sequentially with a single statement). Consider the code

```
1  class Date{
2  private:
3    int day; int month; int year;
4  public:
5    Date& setDate(int d, int m, int y){ //returns a reference to Date object
6      setDay(d);
7      setMonth(m);
8      setYear(y);
9      return *this; //enables cascading
10   }
11   Date& setDay(int d){
12     day = d;
13     return *this;
14   }
15   Date& setMonth(int m){
16     if (m >= 1 && m <= 12) //you could validate day too
17       month = m;
18     else
19       throw invalid_argument("Invalid month!");
20     return *this;
21   }
22   Date& setYear(int y){
23     year = y;
24     return *this;
25   }
26   int getDay(){return day;}
27   int getMonth(){return month;}
28   int getYear(){return year;}
29   string toString(){
30     ostringstream out;
31     out << getDay() << "/" << getMonth() << "/" << getYear() << endl;
32     return out.str();
33   }
```

```
34 };
```

Listing 5: Using this to enable cascaded function calls

The code in Listing 6 shows how the cascading calls can be done

```
1  Date d;
2  d.setYear(2018).setDay(17).setMonth(3);
3  cout << "Date is " << d.toString();
```

Listing 6: Using this to enable cascaded function calls

where line 2 shows how calls to 3 functions are cascaded. The output of this code is shown in Figure **??**

```
Date is 17/3/2018

--------------------------------
Process exited after 0.04636 seconds with return value 0
Press any key to continue . . .
```

Figure 2: Output of code in Listing 6

3 friend classes and friend functions

A *friend* function of a class is a non-member function that has the right to access *public* and *private* members of a class.

To declare a non-member function as a *friend*, place the function prototype in the class definition and precede it with the *friend* keyword as line 8 of Listing 8.

```
1  {
2  private:
3    int a = 10;
4    int b = 20;
5  public:
6    int getA() { return a; }
7    friend void accessAA(AA&); //friend function
8    friend int returnA(AA&);
9    friend class BB;
10 };
```

Listing 7: Friend functions

This declares a function accessAA() which receives an address of AA object and can now use this access members of AA from outside AA as

```
1  class BB
2  {
3  private:
4    int a = 300;
5    int b = 400;
6  public:
7    void accessAA(AA& aa){ //since accessAA is a friend function, it can access members of AA
8      cout << "In AA, a =" << aa.getA() << endl;
9    }
10   void showA(){ //this is not a friend function to AA, so can only access members of BB
11     cout << "in BB, a =" << a << endl;
12   }
13   int returnA(AA& aa){
14     return aa.getA(); //Accessing AA function from BB
15   }
```

```
};
```

Listing 8: Friend functions: Accessing members of AA from BB

A *friend* class is a declared as a friend of another class, and it can access members of the class declaring the friend as 7 in Listing 11 declares class B to be a friend of class A.

```
class A {
private:
  int a;
public:
  A() { a = 100; }
  int getA() { return a; }
  friend class B;      // Friend Class
};
```

Listing 9: Friend class

Class B can access class A members as

```
class B {
private:
  int a = 20;
public:
  void showA(A& x) {
    // Since B is friend of A, it can access
    // private members of A. It can also access members of class it sits in
    cout << "In A, a=" << x.a << endl;
    cout << "In B, a=" << a << endl;
  }
};
```

Listing 10: Friend class

NB: friends are not affected by access modifiers. You can declare friends anywhere in you class The main function to test class friendship

```
  A a;
  B b;
  b.showA(a);
  BB bb;
  AA aa;
  bb.accessAA(aa);
  bb.showA();
  cout << "Accessing AA function from BB : " << bb.returnA(aa) << endl;
```

Listing 11: Testing Friends

3.1 Friendship rules

1. Friendship is granted, not taken – For class B to be a friend of class A, class A must explicitly declare that class B it its friend.

2. Friendship is not symmetric – If class B is a friend of class A, you cannot infer that class A is a friend of B.

3. Friendship is not transitive – If class A is a friend of class B, and class B is a friend of class C, you cannot infer that class A is a friend of class C.

4 static Class Members

There is an important exception to the rule that each object of a class has its own copy of data members of a class. In some cases, only one copy of a variable should be shared by all objects of a class.

A static data member is shared by all instances of a class and is not specific to any one object of the class.

a class static data member has a class scope and must be initialized exactly once.

Class private and protected static members can be accessed through class public member functions and friends. A class static member exist even when no object of the class exist. To access a public static member when no objects of a class exist, prefix class name and scope resolution operator (::) to the name of the data member.

```
using namespace std;
class Date
{
  private:
    int day, month, year;
    static const int months = 12;
    static const int daysPerMonth[months];
    static int count;
  public:
    Date(){count++;}
    Date(int d, int m, int y): day{d}, month{m}, year{y}{count++;}
    int getYear(){return year;}
    int getMonth(){return month;}
    int getDay(){return day;}
    void setDay(int d){
      day = checkDay(d);}
    void setMonth(int m){
      if(m > 0 && m < months)
        month = m;
      else
        throw invalid_argument("Invalid month");
    }
    void setYear(int y){year = y;}
    int checkDay(int d){
      try{ //Lets try to set the day
        if (d > 0 && d <= 29 && getMonth() == 2 && leap() )
          return d;
        if (d > 0 && d <= daysPerMonth[getMonth() - 1])
          return d;
      }
      catch(invalid_argument& ia) { //Lets catch an invalid_argument exception here
        cerr << "Invalid days for month "<< getMonth() << "\n"<< ia.what() << endl;
      }
    }
    bool leap(){
      if (getYear() % 400 == 0 || getYear() % 4 == 0 && getYear() % 100 != 0)
        return true;
      else
        return false;
    }
    string toString(){
      ostringstream out;
      out << getDay() << "/" << getMonth() << "/" << getYear() << endl;
      return out.str();
    }
    int getCount(){return count;}
```

```
47      int getMonths(){return months;}
48  };
49  const int Date::daysPerMonth[] = {31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};
50  int Date::count = 0;
```

Listing 12: static class members

And the main function to test this is

```
1  {
2    Date d;
3    Date dd (12, 4, 1997);
4    d.setMonth(2);
5    d.setYear(2015);
6    d.setDay(28);
7    cout << "Number of objects = " << d.getCount() << endl;
8    cout <<"Date 1 : " << d.toString();
9    cout <<"Date 2 : " << dd.toString();
10   cout <<"Numner of months in Date 1 : " << d.getMonths() << endl;
11   cout <<"Numner of months in Date 2 : " << dd.getMonths() << endl;
```

Listing 13: static class members

In this example code, daysPerMonth and months are static variable shared across all objects of the class Date. Output of the main above is

```
Number of objects = 2
Date 1 : 28/2/2015
Date 2 : 12/4/1997
Numner of months in Date 1 : 12
Numner of months in Date 2 : 12

--------------------------------
Process exited after 0.04585 seconds with return value 0
Press any key to continue . . .
```

Figure 3: Output of code in Listing 12 and 13

From the output, you can notice that object **d** and object **dd** both share *months*, *count* and *daysPerMonth* as these three are static members. Notice that *months* is the same (12) for both **d** and **dd** but values of *day*, *month* and *year* are different because each object maintains its own copy of these data members.