



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.

```
1 class Name
2 {
3 private:
4     string firstName = "uRaphie";
5     string lastName = "Test";
6     string salutation = "Mr";
7 public:
8     Name(){}
9     Name(string fn)
10    {
11        firstName = fn;
12    }
13    void setLastName(string ln){ lastName = ln; }
14    string getLastName(){ return lastName; }
15    string toString()
16    {
17        ostringstream out;
18        out << salutation << ". " << lastName << " " << firstName << endl;
19        return out.str();
20    }
21 };
22 class Date
23 {
24 private:
25     int day;
26     int month;
27     int year;
28 public:
29     Date(){}
30     Date(int d, int m, int y) : day{ d }, month{ m }, year{ y }{}
31     void setDay(int d){ day = d; }
32     int getDay(){ return day; }
33     string toString()
34     {
35         ostringstream out;
36         out << day << "/" << month << "/" << year << endl;
37         return out.str();
38     }
```

```
39 };
```

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

```
1 class Person
2 {
3 private:
4     Name name; //name is an object of class Name
5     Date dob; //dob is an object of class Date
6 public:
7     Person(){}
8     Person(Name n) : name{ n }{}
9     Person(Name n, Date d) : name{ n }, dob{ d }{}
10    void setName(Name n){ name = n; }
11    Name getName(){ return name; }
12    void setDoB(Date d){ dob = d; }
13    Date getDoB(){ return dob; }
14    string toString()
15    {
16        ostringstream out;
17        out << "Name : " << name.toString() << "Date of birth : " << dob.toString();
18        return out.str();
19    }
20 };
```

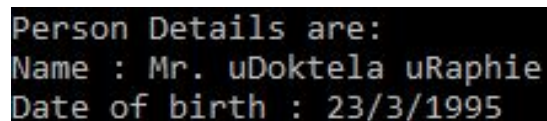
Listing 2: Person as composition of Name and Date

Listing 3 show the main function to test our classes

```
1 int main()
2 {
3     Person p;
4     Name name;
5     Date date(23, 3, 1995);
6     name.setLastName("uDoktela");
7     p.setName(name);
8     p.setDoB(date);
9     cout << "Person Details are: \n" << p.toString();
10    cin.get();
11    return 0;
12 }
```

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    }
```

```
34 };
```

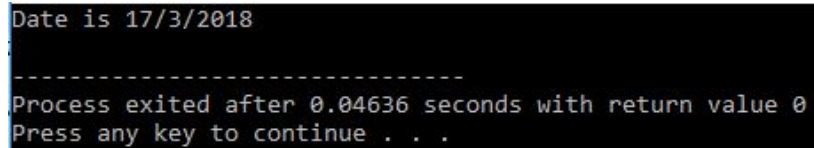
Listing 5: Using this to enable cascaded function calls

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    }
```

```
16 };
```

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

A *friend* class is 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.

```
1 class A {
2 private:
3     int a;
4 public:
5     A() { a = 100; }
6     int getA() { return a; }
7     friend class B;    // Friend Class
8 };
```

Listing 9: Friend class

Class B can access class A members as

```
1 class B {
2 private:
3     int a = 20;
4 public:
5     void showA(A& x) {
6         // Since B is friend of A, it can access
7         // private members of A. It can also access members of class it sits in
8         cout << "In A, a=" << x.a << endl;
9         cout << "In B, a=" << a << endl;
10    }
11 };
```

Listing 10: Friend class

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

```
1 A a;
2 B b;
3 b.showA(a);
4 BB bb;
5 AA aa;
6 bb.accessAA(aa);
7 bb.showA();
8 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 is 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.

```
1 using namespace std;
2 class Date
3 {
4     private:
5         int day, month, year;
6         static const int months = 12;
7         static const int daysPerMonth[months];
8         static int count;
9     public:
10        Date(){count++;}
11        Date(int d, int m, int y): day{d}, month{m}, year{y}{count++;}
12        int getYear(){return year;}
13        int getMonth(){return month;}
14        int getDay(){return day;}
15        void setDay(int d){
16            day = checkDay(d);}
17        void setMonth(int m){
18            if(m > 0 && m < months)
19                month = m;
20            else
21                throw invalid_argument("Invalid month");
22        }
23        void setYear(int y){year = y;}
24        int checkDay(int d){
25            try{ //Lets try to set the day
26                if (d > 0 && d <= 29 && getMonth() == 2 && leap() )
27                    return d;
28                if (d > 0 && d <= daysPerMonth[getMonth() - 1])
29                    return d;
30            }
31            catch(invalid_argument& ia) { //Lets catch an invalid_argument exception here
32                cerr << "Invalid days for month " << getMonth() << "\n" << ia.what() << endl;
33            }
34        }
35        bool leap(){
36            if (getYear() % 400 == 0 || getYear() % 4 == 0 && getYear() % 100 != 0)
37                return true;
38            else
39                return false;
40        }
41        string toString(){
42            ostringstream out;
43            out << getDay() << "/" << getMonth() << "/" << getYear() << endl;
44            return out.str();
45        }
46        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

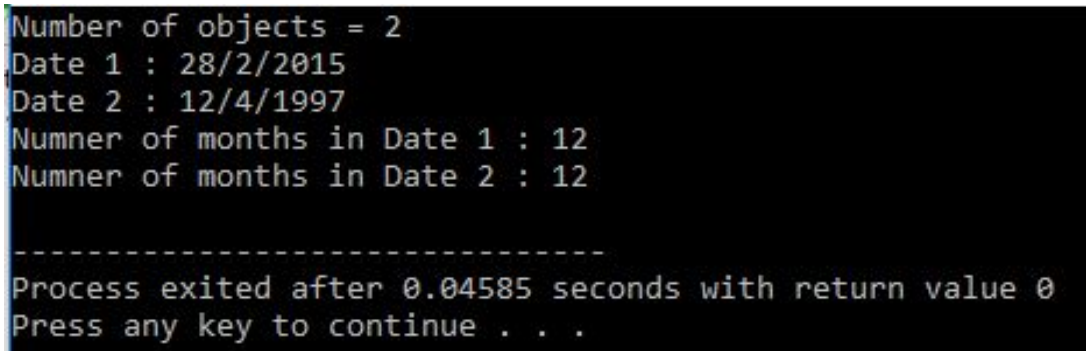
```

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.