Lecture 21:
Classes: A Deeper Look

Ioan Raicu
Department of Electrical Engineering & Computer Science
Northwestern University

EECS 211
Fundamentals of Computer Programming II
May 3rd, 2010

10.3 Composition: Objects as Members of Classes

- Composition
  - Sometimes referred to as a has-a relationship
  - A class can have objects of other classes as members
- An object's constructor can pass arguments to member-object constructors via member initializers.

Software Engineering Observation 10.5

A common form of software reusability is composition, in which a class has objects of other classes as members.

Software Engineering Observation 10.6

*Member objects are constructed in the order in which they're declared in the class definition (not in the order they're listed in the constructor's member initializer list) and before their enclosing class objects (sometimes called **host objects**) are constructed.*


```cpp
// Fig. 10.10: Date.h
// Date class definition; Member functions defined in Date.cpp
#ifndef DATE_H
#define DATE_H

class Date
{
public:
   static const int monthsPerYear = 12; // number of months in a year
   Date( int = 1, int = 1, int = 1900 ); // default constructor
   void print() const; // print date in month/day/year format
   ~Date(); // provided to confirm destruction order
private:
   int month; // 1-12 (January-December)
   int day; // 1-31 based on month
   int year; // any year

   // utility function to check if day is proper for month and year
   int checkDay( int ) const;
}; // end class Date

#endif
```

Fig. 10.10 | Date class definition.

```
// Fig. 10.11: Date.cpp
// Date class member-function definitions.
#include <iostream>
#include "Date.h" // include Date class definition
using namespace std;

// constructor confirms proper value for month; calls
// utility function checkDay to confirm proper value for day
Date::Date( int mn, int dy, int yr )
{
   if ( mn > 0 && mn <= monthsPerYear ) // validate the month
      month = mn;
   else
   {
      month = 1; // invalid month set to 1
      cout << "Invalid month (" << mn << ") set to 1.\n";
   } // end else

   year = yr; // could validate yr
   day = checkDay( dy ); // validate the day

   // output Date object to show when its constructor is called
   cout << "Date object constructor for date ";
```

Fig. 10.11 | Date class member-function definitions. (Part 1 of 3.)

```
   print();
   cout << endl;
} // end Date constructor

// print Date object in form month/day/year
void Date::print() const
{
   cout << month << '/' << day << '/' << year;
} // end function print

// output Date object to show when its destructor is called
Date::~Date()
{
   cout << "Date object destructor for date ";
   print();
   cout << endl;
} // end ~Date destructor

// utility function to confirm proper day value based on
// month and year; handles leap years, too
int Date::checkDay( int testDay ) const
{
   static const int daysPerMonth[ monthsPerYear + 1 ] =
      { 0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 };
```

Fig. 10.11 | Date class member-function definitions. (Part 2 of 3.)

```
   // determine whether testDay is valid for specified month
   if ( testDay > 0 && testDay <= daysPerMonth[ month ] )
      return testDay;

   // February 29 check for leap year
   if ( month == 2 && testDay == 29 && ( year % 400 == 0 ||
      ( year % 4 == 0 && year % 100 != 0 ) ) )
      return testDay;

   cout << "Invalid day (" << testDay << ") set to 1.\n";
   return 1; // leave object in consistent state if bad value
} // end function checkDay
```

Fig. 10.11 | Date class member-function definitions. (Part 3 of 3.)

```cpp
// Fig. 10.12: Employee.h
// Employee class definition showing composition.
// Member functions defined in Employee.cpp.
#ifndef EMPLOYEE_H
#define EMPLOYEE_H

#include <string>
#include "Date.h" // include Date class definition
using namespace std;

class Employee
{
public:
   Employee( const string &, const string &,
      const Date &, const Date & );
   void print() const;
   ~Employee(); // provided to confirm destruction order
private:
   string firstName; // composition: member object
   string lastName; // composition: member object
   const Date birthDate; // composition: member object
   const Date hireDate; // composition: member object
}; // end class Employee

#endif
```

Fig. 10.12 | Employee class definition showing composition.

```cpp
// Fig. 10.13: Employee.cpp
// Employee class member-function definitions.
#include <iostream>
#include "Employee.h" // Employee class definition
#include "Date.h" // Date class definition
using namespace std;

// constructor uses member initializer list to pass initializer
// values to constructors of member objects
Employee::Employee( const string &first, const string &last,
   const Date &dateOfBirth, const Date &dateOfHire )
   : firstName( first ), // initialize firstName
     lastName( last ), // initialize lastName
     birthDate( dateOfBirth ), // initialize birthDate
     hireDate( dateOfHire ) // initialize hireDate
{
   // output Employee object to show when constructor is called
   cout << "Employee object constructor: "
      << firstName << ' ' << lastName << endl;
} // end Employee constructor

```

Fig. 10.13 | Employee class member-function definitions, including constructor with a member initializer list. (Part 1 of 2.)

```cpp
// print Employee object
void Employee::print() const
{
   cout << lastName << ", " << firstName << "  Hired: ";
   hireDate.print();
   cout << "  Birthday: ";
   birthDate.print();
   cout << endl;
} // end function print

// output Employee object to show when its destructor is called
Employee::~Employee()
{
   cout << "Employee object destructor: "
      << lastName << ", " << firstName << endl;
} // end ~Employee destructor
```

Fig. 10.13 | Employee class member-function definitions, including constructor with a member initializer list. (Part 2 of 2.)

10.3 Composition: Objects as Members of Classes (cont.)

- As you study class `Date` (Fig. 10.10), notice that the class does not provide a constructor that receives a parameter of type `Date`.
- Why can the `Employee` constructor's member initializer list initialize the `birthDate` and `hireDate` objects by passing `Date` object's to their `Date` constructors?
- The compiler provides each class with a default copy constructor that copies each data member of the constructor's argument object into the corresponding member of the object being initialized.

```cpp
// Fig. 10.14: fig10_14.cpp
// Demonstrating composition--an object with member objects.
#include <iostream>
#include "Employee.h" // Employee class definition
using namespace std;

int main()
{
   Date birth( 7, 24, 1949 );
   Date hire( 3, 12, 1988 );
   Employee manager( "Bob", "Blue", birth, hire );

   cout << endl;
   manager.print();

   cout << "\nTest Date constructor with invalid values:\n";
   Date lastDayOff( 14, 35, 1994 ); // invalid month and day
   cout << endl;
} // end main
```

Fig. 10.14 | Demonstrating composition—an object with member objects. (Part 1 of 2.)

```
Date object constructor for date 7/24/1949
Date object constructor for date 3/12/1988
Employee object constructor: Bob Blue

Blue, Bob  Hired: 3/12/1988  Birthday: 7/24/1949

Test Date constructor with invalid values:
Invalid month (14) set to 1.
Invalid day (35) set to 1.
Date object constructor for date 1/1/1994

Date object destructor for date 1/1/1994
Employee object destructor: Blue, Bob
Date object destructor for date 3/12/1988
Date object destructor for date 7/24/1949
Date object destructor for date 3/12/1988
Date object destructor for date 7/24/1949
```

There are actually five constructor calls when an **Employee** is constructed—two calls to the **string** class's constructor (lines 12–13 of Fig. 10.13), two calls to the **Date** class's default copy constructor (lines 14–15 of Fig. 10.13) and the call to the **Employee** class's constructor.

Fig. 10.14 | Demonstrating composition—an object with member objects. (Part 2 of 2.)

10.3 Composition: Objects as Members of Classes (cont.)

- If a member object is not initialized through a member initializer, the member object’s default constructor will be called implicitly.
- Values, if any, established by the default constructor can be overrid-den by *set functions*.
- However, for complex initialization, this approach may require significant additional work and time.

Common Programming Error 10.6

A compilation error occurs if a member object is not initialized with a member initializer and the member object's class does not provide a default constructor (i.e., the member object's class defines one or more constructors, but none is a default constructor).

Performance Tip 10.2

Initialize member objects explicitly through member initializers. This eliminates the overhead of "doubly initializing" member objects—once when the member object's default constructor is called and again when set *functions are called in the constructor body (or later) to initialize the member object.*

Software Engineering Observation 10.7

If a class member is an object of another class, making that member object `public` *does not violate the encapsulation and hiding of that member object's* `private` *members. But, it does violate the encapsulation and hiding of the containing class's implementation, so member objects of class types should still be* `private`*, like all other data members.*

10.4 `friend` Functions and `friend` Classes

- A `friend` function of a class is defined outside that class's scope, yet has the right to access the non-`public` (and `public`) members of the class.
- Standalone functions, entire classes or member functions of other classes may be declared to be friends of another class.
- Using `friend` functions can enhance performance.
- Friendship is granted, not taken.
- The friendship relation is neither symmetric nor transitive.

Software Engineering Observation 10.8

Even though the prototypes for `friend` functions appear in the class definition, friends are not member functions.

Software Engineering Observation 10.9

Member access notions of `private`*,* `protected` *and* `public` *are not relevant to* `friend` *declarations, so* `friend` *declarations can be placed anywhere in a class definition.*

Good Programming Practice 10.1

Place all friendship declarations first inside the class definition's body and do not precede them with any access specifier.

Software Engineering Observation 10.10

Some people in the OOP community feel that "friendship" corrupts information hiding and weakens the value of the object-oriented design approach. In this text, we identify several examples of the responsible use of friendship.

```
// Fig. 10.15: fig10_15.cpp
// Friends can access private members of a class.
#include <iostream>
using namespace std;

// Count class definition
class Count
{
   friend void setX( Count &, int ); // friend declaration
public:
   // constructor
   Count()
      : x( 0 ) // initialize x to 0
   {
      // empty body
   } // end constructor Count

   // output x
   void print() const
   {
      cout << x << endl;
   } // end function print
```

Fig. 10.15 | Friends can access `private` members of a class. (Part 1 of 3.)

```cpp
private:
   int x; // data member
}; // end class Count

// function setX can modify private data of Count
// because setX is declared as a friend of Count (line 9)
void setX( Count &c, int val )
{
   c.x = val; // allowed because setX is a friend of Count
} // end function setX

int main()
{
   Count counter; // create Count object

   cout << "counter.x after instantiation: ";
   counter.print();

   setX( counter, 8 ); // set x using a friend function
   cout << "counter.x after call to setX friend function: ";
   counter.print();
} // end main
```

Fig. 10.15 | Friends can access `private` members of a class. (Part 2 of 3.)

```
counter.x after instantiation: 0
counter.x after call to setX friend function: 8
```

Fig. 10.15 | `Friends` can access `private` members of a class. (Part 3 of 3.)

10.5 Using the `this` Pointer

- How do member functions know *which* object's data members to manipulate? Every object has access to its own address through a pointer called `this` (a C++ keyword).
- The `this` pointer is not part of the object itself.
  - The `this` pointer is passed (by the compiler) as an implicit argument to each of the object's non-`static` member functions.
- Objects use the `this` pointer implicitly or explicitly to reference their data members and member functions.
- The type of the `this` pointer depends on the type of the object and whether the member function in which `this` is used is declared `const`.

```cpp
// Fig. 10.16: fig10_16.cpp
// Using the this pointer to refer to object members.
#include <iostream>
using namespace std;

class Test
{
public:
   Test( int = 0 ); // default constructor
   void print() const;
private:
   int x;
}; // end class Test

// constructor
Test::Test( int value )
   : x( value ) // initialize x to value
{
   // empty body
} // end constructor Test
```

Fig. 10.16 | this pointer implicitly and explicitly accessing an object's members. (Part 1 of 3.)

```

// print x using implicit and explicit this pointers;
// the parentheses around *this are required
void Test::print() const
{
   // implicitly use the this pointer to access the member x
   cout << "        x = " << x;

   // explicitly use the this pointer and the arrow operator
   // to access the member x
   cout << "\n  this->x = " << this->x;

   // explicitly use the dereferenced this pointer and
   // the dot operator to access the member x
   cout << "\n(*this).x = " << ( *this ).x << endl;
} // end function print

int main()
{
   Test testObject( 12 ); // instantiate and initialize testObject

   testObject.print();
} // end main
```

Fig. 10.16 | `this` pointer implicitly and explicitly accessing an object's members. (Part 2 of 3.)

```
        x = 12
  this->x = 12
(*this).x = 12
```

Fig. 10.16 | `this` pointer implicitly and explicitly accessing an object's members. (Part 3 of 3.)

Common Programming Error 10.7

Attempting to use the member selection operator (`.`*) with a pointer to an object is a compilation error—the dot member selection operator may be used only with an* lvalue *such as an object's name, a reference to an object or a dereferenced pointer to an object.*

10.5 Using the `this` Pointer (cont.)

- Another use of the `this` pointer is to enable cascaded member-function calls
  - invoking multiple functions in the same statement
- The program of Figs. 10.17–10.19 modifies class `Time`'s *set functions* ***`setTime`****,* ***`setHour`****,* ***`set-Minute`*** *and* ***`setSecond`*** *such that each returns a reference to a* ***`Time`*** *object to enable cascaded member-function calls.*

```cpp
// Fig. 10.17: Time.h
// Cascading member function calls.

// Time class definition.
// Member functions defined in Time.cpp.
#ifndef TIME_H
#define TIME_H

class Time
{
public:
   Time( int = 0, int = 0, int = 0 ); // default constructor

   // set functions (the Time & return types enable cascading)
   Time &setTime( int, int, int ); // set hour, minute, second
   Time &setHour( int ); // set hour
   Time &setMinute( int ); // set minute
   Time &setSecond( int ); // set second

   // get functions (normally declared const)
   int getHour() const; // return hour
   int getMinute() const; // return minute
   int getSecond() const; // return second
```

Fig. 10.17 | `Time` class definition modified to enable cascaded member-function calls. (Part 1 of 2.)

```

   // print functions (normally declared const)
   void printUniversal() const; // print universal time
   void printStandard() const; // print standard time
private:
   int hour; // 0 - 23 (24-hour clock format)
   int minute; // 0 - 59
   int second; // 0 - 59
}; // end class Time

#endif
```

Fig. 10.17 | `Time` class definition modified to enable cascaded member-function calls. (Part 2 of 2.)

```cpp
// Fig. 10.18: Time.cpp
// Time class member-function definitions.
#include <iostream>
#include <iomanip>
#include "Time.h" // Time class definition
using namespace std;

// constructor function to initialize private data;
// calls member function setTime to set variables;
// default values are 0 (see class definition)
Time::Time( int hr, int min, int sec )
{
   setTime( hr, min, sec );
} // end Time constructor

// set values of hour, minute, and second
Time &Time::setTime( int h, int m, int s ) // note Time & return
{
   setHour( h );
   setMinute( m );
   setSecond( s );
   return *this; // enables cascading
} // end function setTime
```

Fig. 10.18 | Time class member-function definitions modified to enable cascaded member-function calls. (Part 1 of 4.)

```
// set hour value
Time &Time::setHour( int h ) // note Time & return
{
   hour = ( h >= 0 && h < 24 ) ? h : 0; // validate hour
   return *this; // enables cascading
} // end function setHour

// set minute value
Time &Time::setMinute( int m ) // note Time & return
{
   minute = ( m >= 0 && m < 60 ) ? m : 0; // validate minute
   return *this; // enables cascading
} // end function setMinute

// set second value
Time &Time::setSecond( int s ) // note Time & return
{
   second = ( s >= 0 && s < 60 ) ? s : 0; // validate second
   return *this; // enables cascading
} // end function setSecond

```

Fig. 10.18 | `Time` class member-function definitions modified to enable cascaded member-function calls. (Part 2 of 4.)

```
// get hour value
int Time::getHour() const
{
   return hour;
} // end function getHour

// get minute value
int Time::getMinute() const
{
   return minute;
} // end function getMinute

// get second value
int Time::getSecond() const
{
   return second;
} // end function getSecond

```

Fig. 10.18 | Time class member-function definitions modified to enable cascaded member-function calls. (Part 3 of 4.)

```
// print Time in universal-time format (HH:MM:SS)
void Time::printUniversal() const
{
   cout << setfill( '0' ) << setw( 2 ) << hour << ":"
      << setw( 2 ) << minute << ":" << setw( 2 ) << second;
} // end function printUniversal

// print Time in standard-time format (HH:MM:SS AM or PM)
void Time::printStandard() const
{
   cout << ( ( hour == 0 || hour == 12 ) ? 12 : hour % 12 )
      << ":" << setfill( '0' ) << setw( 2 ) << minute
      << ":" << setw( 2 ) << second << ( hour < 12 ? " AM" : " PM" );
} // end function printStandard
```

Fig. 10.18 | `Time` class member-function definitions modified to enable cascaded member-function calls. (Part 4 of 4.)

```cpp
// Fig. 10.19: fig10_19.cpp
// Cascading member-function calls with the this pointer.
#include <iostream>
#include "Time.h" // Time class definition
using namespace std;

int main()
{
   Time t; // create Time object

   // cascaded function calls
   t.setHour( 18 ).setMinute( 30 ).setSecond( 22 );

   // output time in universal and standard formats
   cout << "Universal time: ";
   t.printUniversal();

   cout << "\nStandard time: ";
   t.printStandard();

   cout << "\n\nNew standard time: ";

```

Fig. 10.19 | Cascading member-function calls with the `this` pointer. (Part 1 of 2.)

```
    // cascaded function calls
    t.setTime( 20, 20, 20 ).printStandard();
    cout << endl;
} // end main
```

```
Universal time: 18:30:22
Standard time: 6:30:22 PM

New standard time: 8:20:20 PM
```

Fig. 10.19 | Cascading member-function calls with the `this` pointer. (Part 2 of 2.)

10.6 `static` Class Members

- In certain cases, only one copy of a variable should be shared by all objects of a class.
- A `static` data member is used for these and other reasons.
- Such a variable represents “class-wide” information.

Performance Tip 10.3

Use `static` *data members to save storage when a single copy of the data for all objects of a class will suffice.*

10.6 static Class Members (cont.)

- Although they may seem like global variables, a class's `static` data members have class scope.
- `static` members can be declared `public`, `private` or `protected`.
- A fundamental-type `static` data member is initialized by default to `0`.
- If you want a different initial value, a `static` data member can be initialized *once*.
- A `static const` data member of `int` or `enum` type can be initialized in its declaration in the class definition.
- All other `static` data members must be defined *at global namespace scope and can be initialized only in those definitions*.
- If a `static` data member is an object of a class that provides a default constructor, the `static` data member need not be initialized because its default constructor will be called.

10.6 static Class Members (cont.)

- A class's `private` and `protected` `static` members are normally accessed through the class's `public` member functions or `friend`s.
- A class's `static` members exist even when no objects of that class exist.
- To access a `public` `static` class member when no objects of the class exist, prefix the class name and the binary scope resolution operator (`::`) to the name of the data member.
- To access a `private` or `protected` `static` class member when no objects of the class exist, provide a `public` static member function and call the function by prefix-ing its name with the class name and binary scope resolution operator.
- A `static` member function is a service of the *class*, *not of a specific object of the class*.

Software Engineering Observation 10.11

A class's `static` *data members and* `static` *member functions exist and can be used even if no objects of that class have been instantiated.*

Common Programming Error 10.8

It's a compilation error to include keyword `static` *in the definition of a* `static` *data member at global namespace scope.*

```cpp
// Fig. 10.20: Employee.h
// Employee class definition with a static data member to
// track the number of Employee objects in memory
#ifndef EMPLOYEE_H
#define EMPLOYEE_H

#include <string>
using namespace std;

class Employee
{
public:
   Employee( const string &, const string & ); // constructor
   ~Employee(); // destructor
   string getFirstName() const; // return first name
   string getLastName() const; // return last name

   // static member function
   static int getCount(); // return number of objects instantiated
private:
   string firstName;
   string lastName;
```

Fig. 10.20 | Employee class definition with a static data member to track the number of Employee objects in memory. (Part 1 of 2.)

```
    // static data
    static int count; // number of objects instantiated
}; // end class Employee

#endif
```

Fig. 10.20 | `Employee` class definition with a `static` data member to track the number of `Employee` objects in memory. (Part 2 of 2.)

```cpp
// Fig. 10.21: Employee.cpp
// Employee class member-function definitions.
#include <iostream>
#include "Employee.h" // Employee class definition
using namespace std;

// define and initialize static data member at global namespace scope
int Employee::count = 0; // cannot include keyword static

// define static member function that returns number of
// Employee objects instantiated (declared static in Employee.h)
int Employee::getCount()
{
   return count;
} // end static function getCount

// constructor initializes non-static data members and
// increments static data member count
Employee::Employee( const string &first, const string &last )
   : firstName( first ), lastName( last )
{
   ++count; // increment static count of employees
```

Fig. 10.21 | `Employee` class member-function definitions. (Part 1 of 2.)

```
   cout << "Employee constructor for " << firstName
      << ' ' << lastName << " called." << endl;
} // end Employee constructor

// destructor deallocates dynamically allocated memory
Employee::~Employee()
{
   cout << "~Employee() called for " << firstName
      << ' ' << lastName << endl;
   --count; // decrement static count of employees
} // end ~Employee destructor

// return first name of employee
string Employee::getFirstName() const
{
   return firstName; // return copy of first name
} // end function getFirstName

// return last name of employee
string Employee::getLastName() const
{
   return lastName; // return copy of last name
} // end function getLastName
```

Fig. 10.21 | `Employee` class member-function definitions. (Part 2 of 2.)

```cpp
// Fig. 10.22: fig10_22.cpp
// static data member tracking the number of objects of a class.
#include <iostream>
#include "Employee.h" // Employee class definition
using namespace std;

int main()
{
   // no objects exist; use class name and binary scope resolution
   // operator to access static member function getCount
   cout << "Number of employees before instantiation of any objects is "
      << Employee::getCount() << endl; // use class name

   // the following scope creates and destroys
   // Employee objects before main terminates
   {
      Employee e1( "Susan", "Baker" );
      Employee e2( "Robert", "Jones" );

      // two objects exist; call static member function getCount again
      // using the class name and the binary scope resolution operator
      cout << "Number of employees after objects are instantiated is "
         << Employee::getCount();
```

Fig. 10.22 | `static` data member tracking the number of objects of a class. (Part 1 of 3.)

```cpp

      cout << "\n\nEmployee 1: "
         << e1.getFirstName() << " " << e1.getLastName()
         << "\nEmployee 2: "
         << e2.getFirstName() << " " << e2.getLastName() << "\n\n";
   } // end nested scope in main

   // no objects exist, so call static member function getCount again
   // using the class name and the binary scope resolution operator
   cout << "\nNumber of employees after objects are deleted is "
      << Employee::getCount() << endl;
} // end main
```

Fig. 10.22 | `static` data member tracking the number of objects of a class. (Part 2 of 3.)

```
Number of employees before instantiation of any objects is 0
Employee constructor for Susan Baker called.
Employee constructor for Robert Jones called.
Number of employees after objects are instantiated is 2

Employee 1: Susan Baker
Employee 2: Robert Jones

~Employee() called for Robert Jones
~Employee() called for Susan Baker

Number of employees after objects are deleted is 0
```

Fig. 10.22 | `static` data member tracking the number of objects of a class. (Part 3 of 3.)

10.6 static Class Members (cont.)

- A member function should be declared `static` if it does not access non-`static` data members or non-`static` member functions of the class.
- A `static` member function does not have a `this` pointer, because `static` data members and `static` member functions exist independently of any ob-jects of a class.
- The `this` pointer must refer to a specific object of the class, and when a `static` member function is called, there might not be any objects of its class in memory.

Common Programming Error 10.9

Using the `this` *pointer in a* `static` *member function is a compilation error.*

Common Programming Error 10.10

Declaring a `static` *member function* `const` *is a compilation error. The* `const` *qualifier indicates that a function cannot modify the contents of the object in which it operates, but* `static` *member functions exist and operate independently of any objects of the class.*

Questions

?