UEE1303
Objective-Oriented Programming

C++_Lecture 05:

Classes and Objects: Advanced Topics (II)

C: How to Program 8th ed.

Agenda

- `const` (Constant) Objects (chapter 17.10)
- Composition: Objects as Member of Classes (chapter 17.11)
- `friend` function and class (chapter 17.12)
- `this` pointer (chapter 17.13)
- `static` data and methods (chapter 17.14)

`const` (Constant) Objects

- Specify an object is not modifiable and any attempt to modify the object should result in a compilation error.

```
const Time noon( 12, 0, 0);
```

- C++ disallows member function calls for `const` objects unless the member functions themselves are also declared `const`.

Time Class

```
// Time.h  (modified)
#ifndef TIME_H
#define TIME_H
class Time
{
public:
    Time( int = 0, int = 0, int = 0); // constructor
    void setTime(int, int, int); // set time

    int getHour() const;
    int getMinute() const;
    ... ...
    void printUniversal() const;
    void printStandard();
private:
    int hour, minute, second;
}; // end class Time
#endif
```

Partial Member Functions of Time Class

```
// Time.cpp  (modified)
#include <iostream>
#include <iomanip>
#include "Time.h"
using namespace std;

Time::Time( int hour, int minute, int second)
{
    setTime(hour, minute, second);
}
void Time::setTime( int hour, int minute, int second)
{
    setHour(hour);
    setMinute(minute);
    setSecond(second);
}
```

Partial Member Functions of Time Class

```
int Time::getHour() const {
    return hour;
};
int Time::getMinute() const {
    return minute;
};
void Time::printUniversal() const {
    cout << setfill('0') << setw(2) << hour << ":"
         << setw(2) << minute << ":" << setw(2) << second;
}
void Time::printStandard()
{
    cout << ( (hour == 0 || hour == 12) ? 12 : hour % 12 )
          << ":" << setfill( '0' ) << setw( 2 ) << minute
          << ":" << setw( 2 ) << second
          << (hour < 12 ? " AM" : " PM");
}
```

Example of const Objects

```
// Fig. 17.16: fig17_16.cpp
#include "Time.h"

int main() {
    Time wakeUp( 6, 45, 0);
    const Time noon( 12, 0, 0);
                            //OBJECT        MEMBER FUNCTION
    wakeUp.setHour(18);     //non-const     non-const

    noon.setHour(12);       //const         non-const

    wakeUp.getHour();       //non-const     const

    noon.getMinute();       //const         const
    noon.printUniversal();  //const         const

    noon.printStandard();   //const         non-const
    return 0;
}
```

Example of `const` Objects (cont.)

- screen output

```
fig18_03.cpp:13: error: passing 'const Time' as 'this' argument of
   'void Time::setHour(int)' discards qualifiers
fig18_03.cpp:20: error: passing 'const Time' as 'this' argument of
   'void Time::printStandard()' discards qualifiers
```

Initializing a `const` Data Member

- All data members can be initialized using member initializer syntax, but
  - `const` data members and
  - data member that are references

  must be initialized using member initializers.

```
Time::Time(int h, int m, int s)
               :hour(h), minute(m), second(s)
{
 // body intentionally empty
}
```

Initializer list

Example of Member Initializer

```
// Fig. 18.4: Increment.h
// Date class definition
#ifndef INCREMENT_H
#define INCREMENT_H
class Increment
{
public:
    Increment( int c = 0, int = 1); // default constructor
    void addIncrement() {
        count += increment;
    }
    void print() const;
private:
    int count;
    const int increment; // const data member
}; // end class Increment
#endif
```

Example of Member Initializer (cont.)

```
// Fig. 18.5: Increment.cpp
#include <iostream>
#include "Increment.h"
using namespace std;

Increment::Increment(int c, int i) // constructor
                           : count (c),   // non-const member
                             increment(i) // const member
{
    // empty body
}
void Increment::print () const {
    cout << "count = " << count << ", increment = "
         << increment << endl;
}
```

Example of Member Initializer (cont.)

```
// Fig. 18.6: fig18_06.cpp
#include <iostream>
#include "Increment.h"
using namespace std;

int main()
{
    Increment value (10, 5);
    cout << "Before incrementing: ";
    value.print();
    for (int j = 1; j <= 3; j++)
    {
        value.addIncrement();
        cout << "After increment " << j << ": ";
        value.print();
    }
    return 0;
}
```

Example of Member Initializer (cont.)

- screen output

```
Before incrementing: count = 10, increment = 5
After increment 1: count = 15, increment = 5
After increment 2: count = 20, increment = 5
After increment 3: count = 25, increment = 5
```

Erroneously Attempting to Initialize a const Data Member with an Assignment

```
// Fig. 18.7: Increment.h
// Date class definition
#ifndef INCREMENT_H
#define INCREMENT_H
class Increment
{
public:
    Increment( int c = 0, int = 1); // default constructor
    void addIncrement() {
        count += increment;
    }
    void print() const;
private:
    int count;
    const int increment; // const data member
}; // end class Increment
#endif
```

Erroneously Attempting to Initialize a `const` Data Member with an Assignment

```
// Fig. 18.8: Increment.cpp
#include <iostream>
#include "Increment.h"
using namespace std;

Increment::Increment(int c, int i) // constructor
{
    count = c;
    increment = i; // ERROR: Cannot modify a const object
}
void Increment::print () const {
    cout << "count = " << count << ", increment = "
         << increment << endl;
}
```

Example of Member Initializer (cont.)

```
// Fig. 18.9: fig18_09.cpp
#include <iostream>
#include "Increment.h"
using namespace std;

int main()
{
    Increment value (10, 5);
    cout << "Before incrementing: ";
    value.print();
    for (int j = 1; j <= 3; j++)
    {
        value.addIncrement();
        cout << "After increment " << j << ": ";
        value.print();
    }
    return 0;
}
```

Example of Member Initializer (cont.)

- screen output

```
Increment.cpp:9: error: uninitialized member 'Increment::increment' with
   'const' type 'const int'
Increment.cpp:12: error: assignment of read-only data-member
   'Increment::increment'
```

Composition: Objects as Members of Classes

- An `AlarmClock` object needs to know when it's supposed to sound its alarm, so why not include a `Time` object as a member of the `AlarmClock` class?

- Composition ⇨ uses an object of one class within the object of another class
  - form a "has-a" relationship
  - top-level class are called *composed classes*
  - contained objects are called *member objects*

Date.h

```
// Fig. 17.17: Date.h
#ifndef DATE_H
#define DATE_H
class Date
{
public:
    static const int monthsPerYear = 12;
    Date( int = 1, int = 1, int = 1900); // constructor
    void print() const;
    ~Date();
private:
    int month;
    int day;
    int year;
    // utility function
    // check if day is proper for month and year
    int checkDay( int ) const;
}; // end class Date
#endif
```

Date.cpp

```
// Fig. 17.18: Date.cpp
#include <iostream>
#include "Date.h"
using namespace std;
Date::Date( int mn, int dy, int yr ) // constructor
{
   if ( mn > 0 && mn <= monthsPerYear ) // validation
      month = mn;
   else
   {
      month = 1; // invalid month set to 1
      cout << "Invalid month (" << mn << ") set to 1.\n";
   }
   year = yr;
   day = checkDay( dy ); // validate the day
   cout << "Date object constructor for date ";
   print();
   cout << endl;
} // end Date constructor
```

Date.cpp (cont.)

```
// 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
```

Date.cpp (cont.)

```
// 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 };
   // determine whether testDay is valid
   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;
} // end function checkDay
```

Employee.h

```
// Fig. 17.19: Employee.h
#ifndef EMPLOYEE_H
#define EMPLOYEE_H
#include <string>
#include "Date.h"
using namespace std;
class Employee {
public:
    Employee( const string &, const string &, const Date &,
const Date &);
    void print() const;
    ~Employee();
private:
    string fisrtName;
    string lastName;
    const Date birthDate;
    const Date hireDate;
}; // end class Employee
#endif
```

Employee.cpp

```
// Fig. 17.20: Employee.cpp
#include <iostream>
#include "Employee.h"
#include "Date.h"
using namespace std;
Employee::Employee( const string &first, const string &
second, const Date &dateOfBirth, const Date &dayOfHire)
             : firstName(first), lastName(last),
               birthDate(dateOfBirth), hireDate(dayOfHire)
{
    cout << "Employee object constructor:"
         << firstName << ' ' << lastName << endl;
} // constructor
Employee::~Employee()
{
    cout << "Employee object destructor:"
         << lastName << ", " << firstName << endl;
} // destructor
```

Employee.cpp (cont.)

```
void Employee::print() const {
    cout << lastName << ", " << firstName << " Hired: ";
    hireDate.print();
    cout << " Birthday: ";
    birthate.print();
    cout << endl;
}
```

Example of Composition

```
// Fig. 17.21: fig17_21.cpp
#include <iostream>
#include "Employee.h"
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
    return 0;
}
```

Example of Composition (cont.)

- screen output

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

Blue, Bob Hire: 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
```

Example of Composition (cont.)

```
Employee::Employee( const string & first, const string
& second, const Date &dateOfBirth, const Date
&dayOfHire)
                : firstName(first), lastName(last),
                  birthDate(dateOfBirth),
                  hireDate(dayOfHire)
{
    cout << "Employee object constructor:"
         << firstName << ' ' << lastName << endl;
} // constructor
```

- When each of the `Employee`'s `Date` member object's is initialized in the `Employee` constructor's member initializer list, the *default copy constructor* for class `Date` is called.

Example 2 of Composition

```
class CPoint {
    int x, y;
public:
    CPoint() { x = 0; y = 0; }
    CPoint(int a, int b) { x = a; y = b; }
    void set(int a, int b) { x = a; y = b; }
    void move(int a, int b) { x += a; y += b; }
};
```

```
class CRect {
    CPoint tl, br;//top-left and bottom-right
public:
    CRect() { tl.set(0,0); br.set(0,0); }
    CRect(int a, int b, int c, int d) {
        tl.set(a,b); br.set(c,d); }
    CRect(const CPoint &p, const CPoint &q) {
        tl = p; br = q; }
};
```

Example 2 of Composition (cont.)

- Can CRect be declared as follows?

```
class CRect {
    CPoint tl(0,0); //top-left point
    CPoint br(0,0); //bottom-right point
    ...
};
```

Both answers are NO!

- Can a CRect constructor be defined as follows?

```
CRect::CRect(int a, int b, int c, int d) {
    tl(a,b); //top-left point
    br(c,d); //bottom-right point
    ...
}
```

Example 2 of Composition (cont.)

- The first solution for legal initialization is

```
    CRect (int a, int b, int c, int d) {
        tl.set(a,b); br.set(c,d); }
    CRect(const CPoint &p, const CPoint &q) {
        tl = p; br = q; }
```

- The second solution creates a new copy constructor in class CPoint

```
//class CPoint
    CPoint(int a, int b) { x = a; y = b; }
    CPoint(const CPoint &old) {
        x = old.x; y = old.y; }
```

```
//class CRect
    CRect(int a, int b, int c, int d)
                          :tl(a,b), br(c,d) {}
    CRect(const CPoint &p, const CPoint &q)
                          :tl(p), br(q) {}
```

`friend` Functions

- Hiding data inside a class and letting only class member functions have direct access to private data is a very import OOP concept
- But C++ also provides another of function to access members in class ⇨ `friend` functions
  - `friend` functions are not member functions but can still *access class private member*
  - *defined outside of class scope*
- Reasons to have `friend` functions:
  - to access private members of two or more different classes
  - for I/O or operator functions

Properties of `friend` Functions

- Properties:
  - placed *inside the class definition* and preceded by the `friend` keyword
  - *defined outside the class* as a normal, non-member function
  - called like a normal non-member function
- If a function is a friend of two or more different classes, each class much contain the friend function prototype within its body
- A friend function cannot be inherited but can be a member of one class.

First Example of `friend` Function

```
// Fig. 17.22: fig17_22.cpp
// Friends can access private members of a class.
#include <iostream>
using namespace std;
class Count
{
    friend void setX( Count &, int );// friend declaration
public:
    Count() // constructor
          : x( 0 ) // initialize x to 0
    {   // empty body    }
    void print() const
    {
        cout << x << endl;
    }
private:
   int x; // data member
}; // end class Count
```

First Example of `friend` Function (cont.)

```
// function setX can modify private data of Count
void setX( Count &c, int val )
{
    c.x = val; // allowed
}
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
```

Second Example of `friend` Function

```
//in CPoint.cpp
class CPoint {
    int x, y;
    friend CPoint offset(CPoint &,int);
public:
    CPoint() { x = 0; y = 0; }
    CPoint(int a, int b) { x = a; y = b; }
    void print() { cout << x << " ";
                   cout << y << endl; }
};
CPoint offset(CPoint & pt, int diff) {
    pt.x += diff; pt.y += diff;
    return pt;
}
```

```
//in main()
CPoint p1( 3, 5 ); p1.print();
offset(p1, 4); p1.print();
```

`friend` Classes

- An entire class can be a friend of another class ⇨ `friend` class
  - can be used when all member functions of one class should access the data of another class
  - require *prototypes* to be placed within each class individually
- An entire class can be designed as `friend`
  - all its member functions automatically granted a friendship by the class
  - but "class `B` is a `friend` of class `A`" does not imply "class `A` is a `friend` of class `B`"

Example of `friend` Class

```
// CPoint.h
#ifndef CPOINT_H
#define CPOINT_H
#include <iostream>
class CPoint {
public:
    CPoint() { x = 0; y = 0; }
    CPoint(int a, int b) { x = a; y = b; }
    void set(int a, int b) {
        x = a; y = b;
    }
    void Print() {
        std::cout << x << " " << y<< std::endl;
    }
private:
    friend class CLine;
    int x, y;
    void Offset(int diff) {
        x += diff; y += diff;
    }
};  // end class CPoint
#endif
```

```cpp
// CLine.cpp
#ifndef CLINE_H
#define CLINE_H
#include <iostream>
#include "CPoint.h"
class CLine {
public:
    CLine(int x, int y, int w, int z) :
          p1(x,y), p2(w,z) {} // or??
    void Print(CPoint p1, CPoint p2) {
        // call public member function in
        // class CPoint for p and q
        std::cout << "Point 1:"; p1.Print();
        std::cout << "Point 2:"; p2.Print();
    }
    void Display() {
        p1.Offset(3); // call a friend function
        p2.Offset(5); // function in CPoint
        Print(p1, p2);
    }
private:
    CPoint p1, p2;
}; // end class CLine
#endif
```

Example of `friend` Class (cont.)

```
// main
#include <iostream>
#include "CPoint.h"
#include "CLine.h"
using namespace std;

int main()
{
    CPoint pi( 2, 4 ), pj( 6, 8 );
    pi.Print();
    pj.Print();
    pi.Offset(3); //error! Why?

    CLine lk( 3, 5, 7, 9 );
    lk.Display();

    return 0;
}
```

`this` Pointer

- How C++ guarantee that the data member is correctly referenced via member function?
- Example:

```
class Box {
public:
    Box(int h, int w, int l):
        hgt(h), wid(w), len(l) {}
    int vol() { return hgt*wid*len; }
private:
    int hgt, wid, len;
};
```

```
Box a(2,4,6), b(3,5,7);
```

- `a.vol()` and `b.vol()` use the same function

`this` Pointer (cont.)

- The C++ compiler creates and uses a special kind of pointer called the `this` pointer.
  - Stores the address of an object used to call a *non-static* member function.
  - typically hidden from programmer
  - handled by the compiler
- Each *non-static* member function name must be preceded with an object name when called the function
  - address of that object is passed to the function and stored in `this`
  - access the data stored in the object by dereferencing `this`, i.e., `this->member`

Example of `this` Pointers

```
class CPoint {
    int x, y;
public:
    //this is hidden in setPt() and Print()
    void setPt(int a, int b) { x = a; y = b; }
    void Print() { cout << x << " "
                        << y << endl; }
};
```

```
class CPoint {
    int x, y;
public:
    //use this to refer the current object
    void setPt(int a, int b) {
        this->x = a; this->y = b; }
    void Print() { cout << this->x << " "
                        << this->y << endl; }
};
```

Example of `this` Pointers (cont.)

```
class CPoint {
    int x, y;
public:
    //use this to refer the current object
    void setPt(int a, int b) {
        (*this).x = a; (*this).y = b; }
    void Print() {
        cout << (*this).x << " "
             << (*this).y << endl; }
};
```

- Pair of parentheses cannot be omitted because dot (`.`) operator precedes dereference(`*`) operator

Another Example

```
// Fig. 17.23: fig17_23.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; // data member
}; // end class Test
Test::Test( int value): x(value)
{
    // empty body
}
```

Another Example (cont.)

```
void Test::print() const
{
    // implicitly use the this pointer
    cout << "        x = " << x;
    // explicitly use the this pointer and the arrow
operator to access member x
    cout << "\n  this->x = " << this->x;
    // explicitly use the deferenced this pointer and the
dot operator to access member x
    cout << "\n(*this).x = " << (*this).x << endl;
}
int main()
{
    Test testObject(12);
    testObject.print();
}
```

Return an Object

- There are cases when it is necessary to return an object that is used to call the function.
- Because the object is passed implicitly by this, it is necessary to use this explicitly in order to return the object, as follows:

```
return *this;
```

Example of Return Objects from Functions

```
#include <iostream>
using namespace std;
class Pixel
{
public:
    Pixel() {x = 0; y = 0;} // constructor
    ~Pixel() { cout << "\t\tPixel destroyed!\n";}
    void setXY(int x1, int y1) { x = x1; y = y1;}
    void getCoord() {
        cout << "Pixel’s coordinates: \n";
        cout << "X = " << x << "Y = " << y << endl;
    }
    Pixel move_10(Pixel t) {
        t.x = t.x + 10;  t.y = t.y + 10;
        return t;
    }
private:
    int x, y;
}; // end class Pixel
```

Example of Return Objects from Functions (cont.)

```
Pixel setCoord() {
    int x1, y1;
    cout << "Enter x and y coordinates =>";
    cin >> x1 >> y1;
    Pixel temp;
    temp.setXY(x1, y1);
    return temp;
}
int main()
{
    Pixel p1, p2;
    p1 = setCoord();
    p1.getCoord();
    p2 = p1.move_10(p1);
    p2.getCoord();
    p1.getCoord();
    return 0;
}
```

Example of Return an Object

```
#include <iostream>
using namespace std;
class Pixel
{
public:
    Pixel() {x = 0; y = 0;} // constructor
    ~Pixel() { cout << "\t\tPixel destroyed!\n";}
    void setCoord(int x1, int y1) { x = x1; y = y1;}
    void getCoord() {
        cout << "Pixel’s coordinates: \n";
        cout << "X = " << x << "Y = " << y << endl;
    }
    Pixel move_10() {
        x = x + 10;  y = y + 10;
        return *this;
    }
private:
    int x, y;
}; // end class Pixel
```

Example of Return an Object (cont.)

```
int main()
{
    Pixel p1, p2;
    int x1, y1;
    cout << "Enter X and Y coordinates => ";
    cin >> x1 >> y1;
    p1.setCoord(x1, y1);
    p1.getCoord();
    p2 = p1.move_10();
    p2.getCoord();
    p1.getCoord();
    return 0;
}
```

Using the `this` Pointer to Enable Cascaded Function Calls

```
// Fig. 17.24: Time.h  (partial code)
#ifndef TIME_H
#define TIME_H
class Time
{
public:
    Time( int = 0, int = 0, int = 0); // constructor
    Time &setTime(int, int, int);
    Time &setHour(int);
    Time &setMinute(int);
    Time &setSecond(int);
    void printUniversal();
    ……
private:
    int hour;   // 0 – 23 (24-hour clock format)
    int minute; // 0 – 59
    int second; // 0 – 59
}; // end class Time
#endif
```

Using the `this` Pointer to Enable Cascaded Function Calls (cont.)

```
// Fig. 17.25: Time.cpp (partial code)
Time &Time::setTime(int h, int m, int s) {
    setHour(h);
    setMinute(m);
    setSecod(s);
    return *this;
}

Time &Time:: setHour(int h) {
    hour = ( h >= 0 && h < 24 )? h : 0;
    return *this;
}
Time &Time:: setMinute(int m) {
    minute = ( m >= 0 && m < 60 )? m : 0;
    return *this;
}
Time &Time:: setSecond(int s) {
    second = ( s >= 0 && s < 60 )? s : 0;
    return *this;
}
```

Using the `this` Pointer to Enable Cascaded Function Calls (cont.)

```
// Fig. 17.26: fig17_26.cpp
#include <iostream>
#include "Time.h"
using namespace std;
int main() {
    Time t;
    // cascaded function calls
    t.setHour(18).setMinute(30).setSecond(22);
    cout << " Universal time: ";
    t.printUniversal();
    cout << "\nStandard time: ";
    t.printStandard();
    cout << "\nNew standard time: ";
    t.setTime(20, 20, 20).printStandard();

    return 0;
}
```

`static` Data Members

- Class provides data encapsulation and hiding
  - but results in difficulties data sharing and external visit
  - can be resolved by global variables
- A better solution ⇨ static members
  - is specific to one class
  - has a scope shared by the objects of the same class
- A static member can be accessed
  (1) with class methods
  (2) outside class methods

```
⟨type⟩ ⟨class⟩::⟨static_data⟩ = ⟨value⟩;
```

Define `static` Data Members

- A static data member is defined by
  - use keyword `static` to declare a data member in class
  - allocate memory and initialize outside class
  - not limited by the access modifiers
- Example:

```
class X {
private:
    //declare a static data variable
    static int count;
};
//initialize the static member
int X::count = 0;
```

`static` Data Members (cont.)

- All `static` data members exist in memory before any object of their class is instantiated.
  - All objects of class "share" one copy
  - One object changes it → all see change
- Being independent from objects, they are good candidates for the following:
  - Counters that count the number of objects instantiated or destroyed.
  - Totals that accumulate values stored in objects.
  - Variables that control access to the devices shared by all objects, such as servers, prints, and disk drives.

`static` Member Functions

- Member functions can be `static` with the `static` keyword in a class definition.
  - A `static` member function is not attached to any object.
  - An object is not needed when calling `static` member functions.

```
<class_name>::<static_function>;
```

  - A `static` member function does not have direct access to the `private` class data members.
    - A pointer of a reference to an object should be passed to the function to enable access to the object's `private` data.
  - A `static` member function does not have a **`this`** pointer.

Example 1 of `static` Members

- use `static` data members from class methods

```
class CNum {
public:
    CNum(int a) { x = a; y += a; }
    static void fun(CNum m) {
        cout << m.x << "vs. " << y << endl; }
private:
    int x;
    static int y;
};
int CNum::y = 0;
```

```
//in main()
CNum O1(4), O2(7);
CNum ::fun(O1);
CNum ::fun(O2);
```

What to display on screen?

Example 2 of `static` Members

```
// Program demonstrates static member functions
#include <iostream>
using namespace std;

class Node {
public:
    static int ncount; // counts nodes
    static int pcount; // counts printer’s usage
    Node(){ num = 1; ncount++; }
    Node(int x) { num = x; ncount++; }
    ~Node() { ncount--; }
    static void printer(Node &n); // static function
private:
    int num;
};  // end class Node
void Node::printer(Node &n) {
    pcount++;
    cout << "\nThere are " << ncount << " nodes." << endl;
    cout << "\t\t\tNode #" << n.num << " uses printer
            now!" << endl;
    cout << pcount << " node(s) used printer so far."
         << endl;
}
```

Example 2 of `static` Members (cont.)

```
int Node::ncount = 0;
int Node::pcount = 0;
int main()
{
    cout << "\nThere are " << Node::ncount << " nodes."
         << endl;
    Node n1, n2(2), n3(3);
    Node::printer(n2);
    {
        Node n4(4), n5(5);
        Node::printer(n5);
    }
    cout << "\nThere are " << Node::ncount << " nodes. "
         << endl;
}
```

Example 2 of `static` Members (cont.)

- screen output

```
There are 0 nodes.

There are 3 nodes.
                        Node #2 uses printer now!
1 node(s) used printer so far.

There are 5 nodes.
                        Node #5 uses printer now!
2 node(s) used printer so far.

There are 3 nodes.
```

Example 3 of `static` Members

```
// Fig. 17.27: 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
```

Example 3 of `static` Members (cont.)

```
private:
    string firstName;
    string lastName;
    // static data
    static int count; // number of objects instantiated
}; // end class Employee

#endif
```

Example 3 of `static` Members (cont.)

```
// Fig. 17.28: 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
```

Example 3 of `static` Members (cont.)

```
// 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
    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
```

Example 3 of `static` Members (cont.)

```
// 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
```

Example 3 of `static` Members (cont.)

```
// Fig. 17.29: fig17_29.cpp
#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 func.
    cout << "Number of employees before instantiation of
             any objects is "
         << Employee::getCount() << endl;

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

Example 3 of `static` Members (cont.)

```
        // two objects exist;
        cout << "Number of employees after objects are
                 instantiated is "
             << Employee::getCount();

        cout << "\n\nEmployee 1: "
             << e1.getFirstName() << " "
             << e1.getLastName()  << "\nEmployee 2: "
             << e2.getFirstName() << " "
             << e2.getLastName() << "\n\n";
    } // end nested scope in main
    // no objects exist
    cout << "\nNumber of employees after objects are
             deleted is "
         << Employee::getCount() << endl;
} // end main
```

Example 3 of `static` Members (cont.)

- screen output

```
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
```

Example 4 of `static` Members

Display 7.6 Static Members

```cpp
#include <iostream>
using namespace std;

class Server
{
public:
    Server(char letterName);
    static int getTurn( );
    void serveOne( );
    static bool stillOpen( );
private:
    static int turn;
    static int lastServed;
    static bool nowOpen;
    char name;
};

int Server:: turn = 0;
int Server:: lastServed = 0;
bool Server::nowOpen = true;
```

Example 4 of `static` Members (cont.)

```cpp
int main( )
{
    Server s1('A'), s2('B');
    int number, count;
    do
    {
        cout << "How many in your group? ";
        cin >> number;
        cout << "Your turns are: ";
        for (count = 0; count < number; count++)
            cout << Server::getTurn( ) << ' ';
        cout << endl;
        s1.serveOne( );
        s2.serveOne( );
    } while (Server::stillOpen( ));

    cout << "Now closing service.\n";

    return 0;
}
```

Example 4 of `static` Members (cont.)

Display 7.6 Static Members

```
Server::Server(char letterName) : name(letterName)
{/*Intentionally empty*/}

int Server::getTurn( )
{
    turn++;
    return turn;
}
bool Server::stillOpen( )
{
    return nowOpen;
}

void Server::serveOne( )
{
    if (nowOpen && lastServed < turn)
    {
        lastServed++;
        cout << "Server " << name
             << " now serving " << lastServed << endl;
     }
```

Since **getTurn** *is static, only static members can be referenced in here.*

Example 4 of `static` Members (cont.)

```
        if (lastServed >= turn) //Everyone served
            nowOpen = false;
    }
```

SAMPLE DIALOGUE

How many in your group? **3**
Your turns are: 1 2 3
Server A now serving 1
Server B now serving 2
How many in your group? **2**
Your turns are: 4 5
Server A now serving 3
Server B now serving 4
How many in your group? **0**
Your turns are:
Server A now serving 5
Now closing service.

Summary

- Composition
  - Composed classes and member objects
  - Legal initialization

- friend
  - When to use a `friend` function?
  - When to use a `friend` class and how?

- Different kinds of members
  - What is a `static` data member?
  - Why do we need `const` methods?

References

- Paul Deitel and Harvey Deitel, "C How to Program" Eighth Edition
  - Chapter 17
- Paul Deitel and Harvey Deitel, "C++ How to Program (late objects version)" Seventh Edition
  - Chapter 9: Class
  - Chapter 10: Classes: A Deeper Look
- W. Savitch, "Absolute C++," Fourth Edition
  - Chapter 7 (7.2), 8(8.2)