計算機程式 OBJECT-ORIENTED PROGRAMMING 物件導向程式設計 DME1584

Lecture #06

Classes Part02

Composition: Objects as Members of Classes

- Composition
  - Class has objects of other classes as members
- Construction of objects
  - Member objects constructed in order declared
    - Not in order of constructor’s member initializer list
    - Constructed before enclosing class objects (host objects)

Outline

date1.h (1 of 1)

```
// Fig. 7.6: date1.h
// Date class definition.
// Member functions defined in date1.cpp
#ifndef DATE1_H
#define DATE1_H

class Date {

public:
   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 test proper day for month and year
   int checkDay( int ) const;

}; // end class Date

#endif
```

Note no constructor with parameter of type **Date**. Recall compiler provides default copy constructor.

Outline

date1.cpp (1 of 3)

```
// Fig. 7.7: date1.cpp
// Member-function definitions for class Date.
#include <iostream>

using std::cout;
using std::endl;

// include Date class definition from date1.h
#include "date1.h"

// 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 <= 12 )  // validate the month
      month = mn;

   else {                     // invalid month set to 1
      month = 1;
      cout << "Month " << mn << " invalid. Set to month 1.\n";
   }

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

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

Outline

date1.cpp (2 of 3)

```
// 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[ 13 ] =
    { 0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 };

 // 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 << "Day " << testDay << " invalid. Set to day 1.\n";

 return 1;  // leave object in consistent state if bad value

} // end function checkDay
```

Outline

date1.cpp (3 of 3)

Outline

employee1.h (1 of 2)

```
// Fig. 7.8: employee1.h
// Employee class definition.
// Member functions defined in employee1.cpp.
#ifndef EMPLOYEE1_H
#define EMPLOYEE1_H

// include Date class definition from date1.h
#include "date1.h"

class Employee {

public:
   Employee(
      const char *, const char *, const Date &, const Date & );

   void print() const;
   ~Employee();  // provided to confirm destruction order

private:
   char firstName[ 25 ];
   char lastName[ 25 ];
   const Date birthDate;  // composition: member object
   const Date hireDate;   // composition: member object

}; // end class Employee
```

Using composition; **`Employee`** object contains **`Date`** objects as data members.

Outline

employee1.h (2 of 2)

```

#endif
```

employee1.cpp (1 of 3)

```
// Fig. 7.9: employee1.cpp
// Member-function definitions for class Employee.
#include <iostream>

using std::cout;
using std::endl;

#include <cstring>      // strcpy and strlen prototypes

#include "employee1.h"  // Employee class definition
#include "date1.h"      // Date class definition

```

Outline

employee1.cpp
(2 of 3)

```
// constructor uses member initializer list to pass initializer
// values to constructors of member objects birthDate and
// hireDate [Note: This invokes the so-called "default copy
// constructor" which the C++ compiler provides implicitly.]
Employee::Employee( const char *first, const char *last,
   const Date &dateOfBirth, const Date &dateOfHire )
   : birthDate( dateOfBirth ),  // initialize birthDate
     hireDate( dateOfHire )     // initialize hireDate
{
   // copy first into firstName and be sure that it fits
   int length = strlen( first );
   length = ( length < 25 ? length : 24 );
   strncpy( firstName, first, length );
   firstName[ length ] = '\0';

   // copy last into lastName and be sure that it fits
   length = strlen( last );
   length = ( length < 25 ? length : 24 );
   strncpy( lastName, last, length );
   lastName[ length ] = '\0';

   // output Employee object to show when constructor is called
   cout << "Employee object constructor: "
        << firstName << ' ' << lastName << endl;

```

Member initializer syntax to initialize **Date** data members **birthDate** and **hireDate**; compiler uses default copy constructor.

Output to show timing of constructors.

Outline

employee1.cpp
(3 of 3)

```
} // end Employee constructor

// print Employee object
void Employee::print() const
{
   cout << lastName << ", " << firstName << "\nHired: ";
   hireDate.print();
   cout << "  Birth date: ";
   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 destructor ~Employee
```

Output to show timing of destructors.

Outline

fig07_10.cpp
(1 of 1)

```cpp
// Fig. 7.10: fig07_10.cpp
// Demonstrating composition--an object with member objects.
#include <iostream>

using std::cout;
using std::endl;

#include "employee1.h"  // Employee class definition

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

   cout << '\n';
   manager.print();

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

   return 0;

} // end main
```

Create **Date** objects to pass to **Employee** constructor.

Outline

_10.cpp
(1 of 1)

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

Jones, Bob
Hired: 3/12/1988  Birth date: 7/24/1949

Test Date constructor with invalid values:
Month 14 invalid. Set to month 1.
Day 35 invalid. Set to day 1.
Date object constructor for date 1/1/1994

Date object destructor for date 1/1/1994
Employee object destructor: Jones, 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
```

Note two additional **Date** objects constructed; no output since default copy constructor used.

Destructor for host object **manager** runs before destructors for member objects **hireDate** and **birthDate**.

Destructor for **Employee**'s member object **hireDate**.

Destructor for **Employee**'s member object **birthDate**.

Destructor for **Date** object **hire**.

Destructor for **Date** object **birth**.

friend Functions and friend Classes

- **friend** function
  - Defined outside class's scope
  - Right to access non-public members
- Declaring **friend**s
  - Function
    - Precede function prototype with keyword **friend**
  - All member functions of class **ClassTwo** as **friend**s of class **ClassOne**
    - Place declaration of form

      ```
      friend class ClassTwo;
      ```

      in **ClassOne** definition

friend Functions and friend Classes

- Properties of friendship
  - Friendship granted, not taken
    - Class **B** **friend** of class **A**
      - Class **A** must explicitly declare class **B** **friend**
  - Not symmetric
    - Class **B** **friend** of class **A**
    - Class **A** not necessarily **friend** of class **B**
  - Not transitive
    - Class **A** **friend** of class **B**
    - Class **B** **friend** of class **C**
    - Class **A** not necessarily **friend** of Class **C**

Outline

fig07_11.cpp
(1 of 3)

```
// Fig. 7.11: fig07_11.cpp
// Friends can access private members of a class.
#include <iostream>

using std::cout;
using std::endl;

// Count class definition
class Count {
   friend void setX( Count &, int ); // friend declaration

public:

   // constructor
   Count()
      : x( 0 )  // initialize x to 0
   {
      // empty body

   } // end Count constructor

```

Precede function prototype with keyword **`friend`**.

Outline

fig07_11.cpp
(2 of 3)

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

   } // end function print

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
void setX( Count &c, int val )
{
   c.x = val;  // legal: setX is a friend of Count

} // end function setX

```

Pass **`Count`** object since C-style, standalone function.

Since **`setX friend`** of **`Count`**, can access and modify **`private`** data member **`x`**.

Outline

fig07_11.cpp
(3 of 3)

fig07_11.cpp
output (1 of 1)

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

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

   setX( counter, 8 ); // set x with a friend

   cout << "counter.x after call to setX friend function: ";
   counter.print();

   return 0;

} // end main
```

Use **friend** function to access and modify **private** data member **x**.

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

Practice Time: Lab5-01

1. Download the file in E3
   Lab005-01.zip → the same file in page 3 - 11
2. Add the five files into one project
   compile and run
3. Change the file name “Employee1.* ”
   into “Character.cpp” and “Character.h”
4. Change the Class name “Employee” to “Character”
   compile and run

19

20

Using the `this` Pointer

- **`this`** pointer
  - Allows object to access own address
  - Not part of object itself
    - Implicit argument to non-**`static`** member function call
  - Implicitly reference member data and functions
  - Type of **`this`** pointer depends on
    - Type of object
    - Whether member function is **`const`**
    - In non-**`const`** member function of **`Employee`**
      - **`this`** has type **`Employee * const`**
        - Constant pointer to non-constant **`Employee`** object
    - In **`const`** member function of **`Employee`**
      - **`this`** has type **`const Employee * const`**
        - Constant pointer to constant **`Employee`** object

Outline

fig07_13.cpp
(1 of 3)

```cpp
// Fig. 7.13: fig07_13.cpp
// Using the this pointer to refer to object members.
#include <iostream>

using std::cout;
using std::endl;

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 Test constructor
```

Outline

fig07_13.cpp

```cpp

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

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

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

} // end function print

int main()
{
   Test testObject( 12 );

   testObject.print();

   return 0;

```

Implicitly use **this** pointer; only specify name of data member (**x**).

Explicitly use **this** pointer with arrow operator.

Explicitly use **this** pointer; dereference **this** pointer first, then use dot operator.

```
} // end main
```

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

Outline

fig07_13.cpp
(3 of 3)

fig07_13.cpp
output (1 of 1)

Using the `this` Pointer

- Cascaded member function calls
  - Multiple functions invoked in same statement
  - Function returns reference pointer to same object
    `{ return *this; }`
  - Other functions operate on that pointer
  - Functions that do not return references must be called last

```
// Fig. 7.14: time6.h
// Cascading member function calls.

// Time class definition.
// Member functions defined in time6.cpp.
#ifndef TIME6_H
#define TIME6_H

class Time {

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

   // set functions
   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

```

Set functions return reference to **`Time`** object to enable cascaded member function calls.

Outline

time6.h (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
```

Outline

time6.h (2 of 2)

Outline

time6.cpp (1 of 5)

```cpp
// Fig. 7.15: time6.cpp
// Member-function definitions for Time class.
#include <iostream>

using std::cout;

#include <iomanip>

using std::setfill;
using std::setw;

#include "time6.h"  // Time class definition

// 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

```

Outline

time6.cpp (2 of 5)

```cpp
// set values of hour, minute, and second
Time &Time::setTime( int h, int m, int s )
{
   setHour( h );
   setMinute( m );
   setSecond( s );

   return *this;  // enables cascading

} // end function setTime

// set hour value
Time &Time::setHour( int h )
{
   hour = ( h >= 0 && h < 24 ) ? h

   return *this;  // enables cascading

} // end function setHour

```

Return **`*this`** as reference to enable cascaded member function calls.

Return **`*this`** as reference to enable cascaded member function calls.

```cpp
// set minute value
Time &Time::setMinute( int m )
{
   minute = ( m >= 0 && m < 60 )

   return *this; // enables cascading

} // end function setMinute

// set second value
Time &Time::setSecond( int s )
{
   second = ( s >= 0 && s < 60 )

   return *this; // enables cascading

} // end function setSecond

// get hour value
int Time::getHour() const
{
   return hour;

} // end function getHour

```

Return **`*this`** as reference to enable cascaded member function calls.

Return **`*this`** as reference to enable cascaded member function calls.

Outline

time6.cpp (3 of 5)

```cpp
// get minute value
int Time::getMinute() const
{
   return minute;

} // end function getMinute

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

} // end function getSecond

// print Time in universal format
void Time::printUniversal() const
{
   cout << setfill( '0' ) << setw( 2 ) << hour << ":"
        << setw( 2 ) << minute << ":"
        << setw( 2 ) << second;

} // end function printUniversal

```

Outline

time6.cpp (4 of 5)

```
// print Time in standard format
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
```

Outline

time6.cpp (5 of 5)

```
// Fig. 7.16: fig07_16.cpp
// Cascading member function calls with the this pointer.
#include <iostream>

using std::cout;
using std::endl;

#include "time6.h"  // Time class definition

int main()
{
   Time t;

   // 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: ";

```

Cascade member function calls; recall dot operator associates from left to right.

Outline

fig07_16.cpp
(1 of 2)

Outline

```
     // cascaded function calls
     t.setTime( 20, 20, 20 ).printStandard();

     cout << endl;

     return 0;

  } // end main
```

Function call to **printStandard** must appear last; **printStandard** does not return reference to **t**.

7_16.cpp
f 2)

7_16.cpp
ut (1 of 1)

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

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

Dynamic Memory Management with Operators `new` and `delete`

- Dynamic memory management
  - Control allocation and deallocation of memory
  - Operators **`new`** and **`delete`**
    - Include standard header **`<new>`**
      - Access to standard version of **`new`**

Dynamic Memory Management with Operators `new` and `delete`

- **`new`**
  - Consider
    ```
    Time *timePtr;
    timePtr = new Time;
    ```
  - **`new`** operator
    - Creates object of proper size for type **`Time`**
      - Error if no space in memory for object
    - Calls default constructor for object
    - Returns pointer of specified type
  - Providing initializers
    ```
    double *ptr = new double( 3.14159 );
    Time *timePtr = new Time( 12, 0, 0 );
    ```
  - Allocating arrays
    ```
    int *gradesArray = new int[ 10 ];
    ```

Dynamic Memory Management with Operators `new` and `delete`

- **`delete`**
  - Destroy dynamically allocated object and free space
  - Consider
    ```
    delete timePtr;
    ```
  - Operator **`delete`**
    - Calls destructor for object
    - Deallocates memory associated with object
      - Memory can be reused to allocate other objects
  - Deallocating arrays
    ```
    delete [] gradesArray;
    ```
      - Deallocates array to which **`gradesArray`** points
    - If pointer to array of objects
        - First calls destructor for each object in array
        - Then deallocates memory

`static` Class Members

- **`static`** class variable
  - "Class-wide" data
    - Property of class, not specific object of class
  - Efficient when single copy of data is enough
    - Only the **`static`** variable has to be updated
  - May seem like global variables, but have class scope
    - Only accessible to objects of same class
  - Initialized exactly once at file scope
  - Exist even if no objects of class exist
  - Can be **`public`**, **`private`** or **`protected`**

`static` Class Members

- Accessing **`static`** class variables
  - Accessible through any object of class
  - **`public static`** variables
    - Can also be accessed using binary scope resolution operator(`::`)

      `Employee::count`
  - **`private static`** variables
    - When no class member objects exist
      - Can only be accessed via **`public static`** member function
      - To call **`public static`** member function combine class name, binary scope resolution operator (**`::`**) and function name

        `Employee::getCount()`

static Class Members

- **static** member functions
  - Cannot access non-**static** data or functions
  - No **this** pointer for **static** functions
    - **static** data members and **static** member functions exist independent of objects

Outline

employee2.h (1 of 2)

```
// Fig. 7.17: employee2.h
// Employee class definition.
#ifndef EMPLOYEE2_H
#define EMPLOYEE2_H

class Employee {

public:
   Employee( const char *, const char * );  // constructor
   ~Employee();                             // destructor
   const char *getFirstName() const;        // return first name
   const char *getLastName() const;         // return last name

   // static member function
   static int getCount();  // return # objects instantiated

private:
   char *firstName;
   char *lastName;

   // static data member
   static int count;  // number of objects instantiated

}; // end class Employee

```

static member function can only access **static** data members and member functions.

static data member is class-wide data.

```
#endif
```

Outline

employee2.h (2 of 2)

employee2.cpp
(1 of 3)

```
// Fig. 7.18: employee2.cpp
// Member-function definitions for class Employee.
#include <iostream>

using std::cout;
using std::endl;

#include <new>          // C++ standard new operator
#include <cstring>      // strcpy and strlen prototypes

#include "employee2.h"  // Employee class definition

// define and initialize static data member
int Employee::count = 0;

// define static member function that returns number of
// Employee objects instantiated
int Employee::getCount()
{
   return count;

} // end static function getCount
```

Initialize **static** data member exactly once at file scope.

static member function accesses **static** data member **count**.

Outline

employee2.cpp
(2 of 3)

```

// constructor dynamically allocates space for
// first and last name and uses strcpy to copy
// first and last names into the object
Employee::Employee( const char *first, const char *last )
{
   firstName = new char[ strlen( first ) + 1 ];
   strcpy( firstName, first );

   lastName = new char[ strlen( last ) + 1 ];
   strcpy( 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;

```

new operator dynamically allocates space.

Use **static** data member to store total **count** of employees.

Outline

employee2.cpp
(3 of 3)

```
   delete [] firstName;  // recapture memory
   delete [] lastName;   // recapture memory

   --count;  // decrement static count of employees

} // end destructor ~Employee

// return first name of employee
const char *Employee::getFirstName() const
{
   // const before return type prevents client from modifying
   // private data; client should copy returned string before
   // destructor deletes storage to prevent undefined pointer
   return firstName;

} // end function getFirstName

// return last name of employee
const char *Employee::getLastName() const
{
   // const before return type prevents client from modifying
   // private data; client should copy returned string before
   // destructor deletes storage to prevent undefined pointer
   return lastName;

} // end function getLastName
```

Operator **delete** deallocates memory.

Use **static** data member to store total **count** of employees.

Outline

fig07_19.cpp
(1 of 2)

```
// Fig. 7.19: fig07_19.cpp
// Driver to test class Employee.
#include <iostream>

using std::cout;
using std::endl;

#include <new>          // C++ standard new operator

#include "employee2.h"  // Employee class definition

int main()
{
   cout << "Number of employees before instantiation is "
        << Employee::getCount() << endl;   // use class name

   Employee *e1Ptr = new Employee( "Susan", "Baker" );
   Employee *e2Ptr = new Employee( "Robert", "Jones" );

   cout << "Number of employees after instantiation is "
        << e1Ptr->getCount();

```

new operator dynamically allocates space.

static member function can be invoked on any object of class.

```
   cout << "\n\nEmployee 1: "
        << e1Ptr->getFirstName()
        << " " << e1Ptr->getLastName()
        << "\nEmployee 2: "
        << e2Ptr->getFirstName()
        << " " << e2Ptr->getLastName() << "\n\n";

   delete e1Ptr;  // recapture memory
   e1Ptr = 0;     // disconnect pointer from free-store space
   delete e2Ptr;  // recapture memory
   e2Ptr = 0;     // disconnect pointer from free-store space

   cout << "Number of employees after deletion is "
        << Employee::getCount() << endl;

   return 0;

} // end main
```

Operator **delete** deallocates memory.

static member function invoked using binary scope resolution operator (no existing class objects).

Outline

fig07_19.cpp
(2 of 2)

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

Employee 1: Susan Baker
Employee 2: Robert Jones

~Employee() called for Susan Baker
~Employee() called for Robert Jones
Number of employees after deletion is 0
```

Outline

fig07_19.cpp
output (1 of 1)

Practice Time: Lab5-2

1. Continue the code in Lab5_1
2. Use "this" pointer to set the Date .
   "Cascaded member function calls" example
   at page 25-33
3. Add members in "Character"
   member Object: Country, City
   member function:
   Set_Country
   Set_City
4. Store 10 persons with
   first name, last name, birthday, Country, City
   show out the above data

End