IS 0020
Program Design and Software Tools
Introduction to C++ Programming

Lecture 5: Classes
(continued)

Feb 1, 2005

`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`**
  - Want to make 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**


```
// 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**.

fig07_11.cpp
(1 of 3)

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
// because setX is de
void setX( Count &c,
{
   c.x = val;  // leg

} // 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 instantiat
   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
```

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


```
// 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
```

fig07_13.cpp
(1 of 3)

Outline

fig07_13.cpp
(3 of 3)

```
// 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;

} // end main
```

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.

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


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

Outline

time6.h (1 of 2)

```
// 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 );

   // set functions
   Time &setTime( int, int, int ); // s
   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 (2 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. 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

```



```
// 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 ) ?

   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.

time6.cpp (3 of 5)

```
// 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.

time6.cpp (4 of 5)

```
// 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 (5 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

fig07_16.cpp
(1 of 2)

```
// 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.

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

    cout << endl;

    return 0;

} // end main
```

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

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

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

Outline

7_16.cpp
f 2)

7_16.cpp
ut (1 of 1)

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>`**
- **`new`**

```
Time *timePtr;
timePtr = new Time;
```

  - 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 ];
```

21

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

22

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

      `Employee::getCount()`
- **`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;   // r

   // static member function
   static int getCount();  // return # obj

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
```

```
// 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

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

// define static member function that retu
// 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.h (2 of 2)

employee2.cpp (1 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 *
{
   firstName = new char[ strlen( first ) + 1 ];
   strcpy( firstName, first );

   lastName = new char[ str
   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

Outline

employee2.cpp
(3 of 3)

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

   --count;  // decrement static count of employees

} // end destructor ~Emp

// return first name of
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
```

allocates

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

Data Abstraction and Information Hiding

- Information hiding
  - Classes hide implementation details from clients
  - Example: stack data structure
    - Data elements added (pushed) onto top
    - Data elements removed (popped) from top
    - Last-in, first-out (LIFO) data structure
    - Client only wants LIFO data structure
      - Does not care how stack implemented
- Data abstraction
  - Describe functionality of class independent of implementation

Data Abstraction and Information Hiding

- Abstract data types (ADTs)
  - Approximations/models of real-world concepts and behaviors
    - **`int`**, **`float`** are models for a numbers
  - Data representation
  - Operations allowed on those data
- C++ extensible
  - Standard data types cannot be changed, but new data types can be created

Proxy Classes

- Proxy class
  - Hide implementation details of another class
  - Knows only **`public`** interface of class being hidden
  - Enables clients to use class's services without giving access to class's implementation
- Forward class declaration
  - Used when class definition only uses pointer to another class
  - Prevents need for including header file
  - Declares class before referencing
  - Format:

```
class ClassToLoad;
```

Outline

implementation.h
(1 of 2)

```
// Fig. 7.20: implementation.h
// Header file for class Implementation

class Implementation {

public:

   // constructor
   Implementation( int v )
      : value( v )  // initialize value with v
   {
      // empty body

   } // end Implementation constructor

   // set value to v
   void setValue( int v )
   {
      value = v;  // should validate v

   } // end function setValue

   // return value
   int getValue() const
   {
      return value;

   } // end function getValue

private:
   int value;

}; // end class Implementation
```

`public` member function.

Outline

interface.h (1 of 1)

```
// Fig. 7.21: interface.h
// Header file for interface.cpp

class Implementation;     // forward class declaration

class Interface {

public:
   Interface( int );
   void setValue( int );  // same public i
   int getValue() const;  // class Impleme
   ~Interface();

private:

   // requires previous forward declara
   Implementation *ptr;

}; // end class Interface
```

Provide same **`public`** interface as class **`Implementation`**; recall **`setValue`** and **`getValue`** only **`public`** member functions.

Pointer to **`Implementation`** object requires forward class declaration.

Outline

interface.cpp
(1 of 2)

```
// Fig. 7.22: interface.cpp
// Definition of class Interface
#include "interface.h"       // Interface class definition
#include "implementation.h"                           nition

// constructor
Interface::Interface( int v
   : ptr ( new Implementation( v ) )  /
{
   // empty body

} // end Interface constructor

// call Implementation's setValue func
void Interface::setValue( int v )
{
   ptr->setValue( v );

} // end function setValue

```

Maintain pointer to underlying **`Implementation`** object.

`rface` includes header file for class **`Implementation`**.

Invoke corresponding function on underlying **`Implementation`** object.

Outline

interface.cpp
(2 of 2)

```
// call Implementation's getValue function
int Interface::getValue() const
{
   return ptr->getValue();

} // end function getValue

// destructor
Interface::~Interface()
{
   delete ptr;

} // end destructor ~Interface
```

Invoke corresponding function on underlying **`Implementation`** object.

Deallocate underlying **`Implementation`** object.

Outline

fig07_23.cpp
(1 of 1)

fig07_23.cpp
output (1 of 1)

```
// Fig. 7.23: fig07_23.cpp
// Hiding a class's private data with a proxy class.
#include <iostream>

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

#include "interface.h"  // Interface class definition

int main()
{
   Interface i( 5 );

   cout << "Interface contains: " << i.getValue()
        << " before setValue" << endl;

   i.setValue( 10 );

   cout << "Interface contains: " << i.getValue()
        << " after setValue" << endl;

   return 0;

} // end main
```

```
Interface contains: 5 before setValue
Interface contains: 10 after setValue
```

Only include proxy class header file.

Create object of proxy class **`Interface`**; note no mention of **`Implementation`** class.

Invoke member functions via proxy class object.