

---

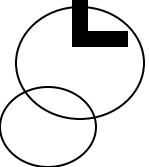
IS 0020  
Program Design and Software Tools  
Introduction to C++ Programming

---

Lecture 4 - Classes

Jan 27, 2004

# friend Functions and friend Classes

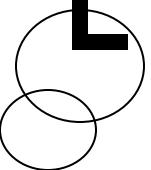


- **friend** function
  - Defined outside class's scope
  - Right to access non-public members
- Declaring **friends**
  - Function
    - Precede function prototype with keyword **friend**
    - All member functions of class **ClassTwo** as **friends** of class **ClassOne**
      - Place declaration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

Precede function prototype  
with keyword **friend**.

```
1 // Fig. 7.11: fig07_11.cpp
2 // Friends can access private members of a class.
3 #include <iostream>
4
5 using std::cout;
6 using std::endl;
7
8 // Count class definition
9 class Count {
10     friend void setX( Count &, int ); // friend declaration
11
12 public:
13
14     // constructor
15     Count()
16         : x( 0 ) // initialize x to 0
17     {
18         // empty body
19
20     } // end Count constructor
21
```



## Outline

fig07\_11.cpp  
(2 of 3)

```
22     // output x
23 void print() const
24 {
25     cout << x << endl;
26
27 } // end function print
28
29 private:
30     int x; // data member
31
32 }; // end class Count
33
34 // function setX can not be a style standalone function.
35 // because setX is de
36 void setX( Count &c,
37 {
38     c.x = val; // leg
39
40 } // end function setX
41
```

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)

```
42 int main()
43 {
44     Count counter;          // create Count object
45
46     cout << "counter.x after instantiation: ";
47     counter.print();
48
49     setX( counter, 8 );    // set x with a friend
50
51     cout << "counter.x after call to setX friend function: ";
52     counter.print();
53
54     return 0;
55
56 } // 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



## Outline

fig07\_13.cpp  
(1 of 3)

```
1 // Fig. 7.13: fig07_13.cpp
2 // Using the this pointer to refer to object members.
3 #include <iostream>
4
5 using std::cout;
6 using std::endl;
7
8 class Test {
9
10 public:
11     Test( int = 0 );      // default constructor
12     void print() const;
13
14 private:
15     int x;
16
17 }; // end class Test
18
19 // constructor
20 Test::Test( int value )
21     : x( value ) // initialize x to value
22 {
23     // empty body
24
25 } // end Test constructor
```



## Outline

07\_13.cpp  
of 3)

```

26
27 // print x using implicit and explicit this pointers;
28 // parentheses around *this required
29 void Test::print() const
30 {
31     // implicitly use this pointer to access member x
32     cout << "        x = " << x;
33
34     // explicitly use this pointer to access member x
35     cout << "\n    this->x = " << this->x;
36
37     // explicitly use dereferenced this pointer and
38     // the dot operator to access member x
39     cout << "\n(*this).x = " << ( *this ).x << endl;
40
41 } // end function print
42
43 int main()
44 {
45     Test testObject( 12 );
46
47     testObject.print();
48
49     return 0;
50

```

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.



## Outline

```
51 } // end main
```

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

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



## Outline

time6.h (1 of 2)

```
1 // Fig. 7.14: time6.h
2 // Cascading member function calls.
3
4 // Time class definition.
5 // Member functions defined in time6.cpp.
6 #ifndef TIME6_H
7 #define TIME6_H
8
9 class Time {
10
11 public:
12     Time( int = 0, int = 0, int = 0 );
13
14     // set functions
15     Time &setTime( int, int, int ); // s
16     Time &setHour( int );        // set hour
17     Time &setMinute( int );     // set minute
18     Time &setSecond( int );    // set second
19
20     // get functions (normally declared const)
21     int getHour() const;       // return hour
22     int getMinute() const;     // return minute
23     int getSecond() const;     // return second
24 }
```

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



## Outline

time6.h (2 of 2)

```
25 // print functions (normally declared const)
26 void printUniversal() const; // print universal time
27 void printStandard() const; // print standard time
28
29 private:
30     int hour; // 0 - 23 (24-hour clock format)
31     int minute; // 0 - 59
32     int second; // 0 - 59
33
34 }; // end class Time
35
36 #endif
```



## Outline

time6.cpp (1 of 5)

```
1 // Fig. 7.15: time6.cpp
2 // Member-function definitions for Time class.
3 #include <iostream>
4
5 using std::cout;
6
7 #include <iomanip>
8
9 using std::setfill;
10 using std::setw;
11
12 #include "time6.h" // Time class definition
13
14 // constructor function to initialize private data;
15 // calls member function setTime to set variables;
16 // default values are 0 (see class definition)
17 Time::Time( int hr, int min, int sec )
18 {
19     setTime( hr, min, sec );
20 }
21 // end Time constructor
22
```



## Outline

time6.cpp (2 of 5)

```
23 // set values of hour, minute, and second
24 Time &Time::setTime( int h, int m, int s )
25 {
26     setHour( h );
27     setMinute( m );
28     setSecond( s );
29
30     return *this; // enables cascaded member calls
31
32 } // end function setTime
33
34 // set hour value
35 Time &Time::setHour( int h )
36 {
37     hour = ( h >= 0 && h < 24 ) ? h : 0;
38
39     return *this; // enables cascaded member calls
40
41 } // end function setHour
42
```

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)

```
43 // set minute value
44 Time &Time::setMinute( int m )
45 {
46     minute = ( m >= 0 && m < 60 )
47
48     return *this; // enables cascading
49
50 } // end function setMinute
51
52 // set second value
53 Time &Time::setSecond( int s )
54 {
55     second = ( s >= 0 && s < 60 )
56
57     return *this; // enables cascading
58
59 } // end function setSecond
60
61 // get hour value
62 int Time::getHour() const
63 {
64     return hour;
65
66 } // end function getHour
67
```

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

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



## Outline

time6.cpp (4 of 5)

```
68 // get minute value
69 int Time::getMinute() const
70 {
71     return minute;
72 }
73 } // end function getMinute
74
75 // get second value
76 int Time::getSecond() const
77 {
78     return second;
79 }
80 } // end function getSecond
81
82 // print Time in universal format
83 void Time::printUniversal() const
84 {
85     cout << setfill( '0' ) << setw( 2 ) << hour << ":"
86         << setw( 2 ) << minute << ":"
87         << setw( 2 ) << second;
88
89 } // end function printUniversal
90
```



## Outline

time6.cpp (5 of 5)

```
91 // print Time in standard format
92 void Time::printStandard() const
93 {
94     cout << ( ( hour == 0 || hour == 12 ) ? 12 : hour % 12 )
95         << ":" << setfill( '0' ) << setw( 2 ) << minute
96         << ":" << setw( 2 ) << second
97         << ( hour < 12 ? " AM" : " PM" );
98
99 } // end function printStandard
```



## Outline

fig07\_16.cpp  
(1 of 2)

```
1 // Fig. 7.16: fig07_16.cpp
2 // Cascading member function calls with the this pointer.
3 #include <iostream>
4
5 using std::cout;
6 using std::endl;
7
8 #include "time6.h" // Time class definition
9
10 int main()
11 {
12     Time t;
13
14     // cascaded function calls
15     t.setHour( 18 ).setMinute( 30 ).setSecond( 22 );
16
17     // output time in universal and standard formats
18     cout << "Universal time: ";
19     t.printUniversal();
20
21     cout << "\nStandard time: ";
22     t.printStandard();
23
24     cout << "\n\nNew standard time: ";
25
```

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

## Outline



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

```
26 // cascaded function calls
27 t.setTime( 20, 20, 20 ).printStandard();
28
29 cout << endl;
30
31 return 0;
32
33 } // end main
```

Universal time: 18:30:22

Standard time: 6:30:22 PM

New standard time: 8:20:20 PM

7\_16.cpp

(2)

7\_16.cpp

out (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**>
    - 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;**

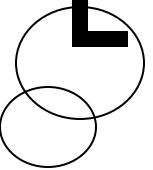
- Deallocates array to which **gradesArray** points

- 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)

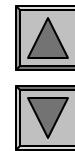
```

1 // Fig. 7.17: employee2.h
2 // Employee class definition.
3 #ifndef EMPLOYEE2_H
4 #define EMPLOYEE2_H
5
6 class Employee {
7
8 public:
9     Employee( const char *, const char * ); // constructor
10    ~Employee(); // destructor
11    const char *getFirstName() const; // return first name
12    const char *getLastName() const; // return last name
13
14    // static member function
15    static int getCount(); // return # obj
16
17 private:
18     char *firstName;
19     char *lastName;
20
21     // static data member
22     static int count; // number of objects instantiated
23
24 }; // end class Employee
25

```

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

**static** data member is  
class-wide data.



## Outline

employee2.h (2 of 2)

employee2.cpp  
(1 of 3)

26 #endif

```

1 // Fig. 7.18: employee2.cpp
2 // Member-function definitions for class Employee.
3 #include <iostream>
4
5 using std::cout;
6 using std::endl;
7
8 #include <new>           // C++ standard new operator
9 #include <cstring>        // strcpy and strlen prototypes
10
11 #include "employee2.h"   // Employee class
12
13 // define and initialize static data member
14 int Employee::count = 0;
15
16 // define static member function that returns
17 // Employee objects instantiated
18 int Employee::getCount()
19 {
20     return count;
21
22 } // end static function getCount

```

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

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



## Outline

Employee2.cpp

```
23
24 // constructor dynamically allocates space for
25 // first and last name and uses strcpy to copy
26 // first and last names into the object
27 Employee::Employee( const char *first, const char *last )
28 {
29     firstName = new char[ strlen( first ) + 1 ];
30     strcpy( firstName, first );
31
32     lastName = new char[ strlen( last ) + 1 ];
33     strcpy( lastName, last );
34
35     ++count; // increment static count of employees
36
37     cout << "Employee constructor for " << firstName
38         << ' ' << lastName << " called." << endl;
39
40 } // end Employee constructor
41
42 // destructor deallocates dynamically allocated memory
43 Employee::~Employee()
44 {
45     cout << "~Employee() called for " << firstName
46         << ' ' << lastName << endl;
47 }
```

**new** operator dynamically allocates space.

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



## Outline

employee2.cpp  
(3 of 3)

```
48     delete [] firstName; // recapture memory
49     delete [] lastName; // recapture memory
50
51     --count; // decrement static count of employees
52
53 } // end destructor ~Emp
54
55 // return first name of employee
56 const char *Employee::getFirstName() const
57 {
58     // const before return type prevents client from modifying
59     // private data; client should copy returned string before
60     // destructor deletes storage to prevent undefined pointer
61     return firstName;
62
63 } // end function getFirstName
64
65 // return last name of employee
66 const char *Employee::getLastName() const
67 {
68     // const before return type prevents client from modifying
69     // private data; client should copy returned string before
70     // destructor deletes storage to prevent undefined pointer
71     return lastName;
72
73 } // end function getLastName
```

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

allocates



## Outline

fig07\_19.cpp  
(1 of 2)

```

1 // Fig. 7.19: fig07_19.cpp
2 // Driver to test class Employee.
3 #include <iostream>
4
5 using std::cout;
6 using std::endl;
7
8 #include <new>           // C++ standard new operator
9
10 #include "employee2.h"   // Employee class definition
11
12 int main()
13 {
14     cout << "Number of employees before instantiation is "
15     << Employee::getCount() << endl;    // use class name
16
17     Employee *e1Ptr = new Employee();
18     Employee *e2Ptr = new Employee();
19
20     cout << "Number of employees aft
21         << e1Ptr->getCount();
```

**new** operator dynamically allocates space.

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



## Outline

fig07\_19.cpp  
(2 of 2)

```
23 cout << "\n\nEmployee 1: "
24     << e1Ptr->getFirstName()
25     << " " << e1Ptr->getLastName()
26     << "\nEmployee 2: "
27     << e2Ptr->getFirstName()
28     << " " << e2Ptr->getLastName() << "\n\n";
29
30 delete e1Ptr; // recapture memory
31 e1Ptr = 0; // disconnect pointer from free-store space
32 delete e2Ptr; // recapture memory
33 e2Ptr = 0; // disconnect pointer f
34
35 cout << "Number of employees a
36     << Employee::getCount() <
```

Operator  
memory

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



## Outline

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

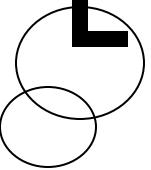
`~Employee()` called for Susan Baker

`~Employee()` called for Robert Jones

Number of employees after deletion is 0

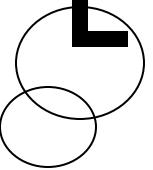
fig07\_19.cpp  
output (1 of 1)

# Data Abstraction and Information Hiding



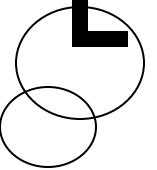
- **Information hiding**
  - Classes hide implementation details from clients
  - Example: stack data structure
    - Data elements added (pushed) onto top
    - Data elements removed (popped) from top
    - Last-in, first-out (LIFO) data structure
    - Client only wants LIFO data structure
      - 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 numbers
  - Data representation
  - Operations allowed on those data
- C++ extensible
  - Standard data types cannot be changed, but new data types can be created

# Example: Array Abstract Data Type



- ADT array
  - Could include
    - Subscript range checking
    - Arbitrary range of subscripts
      - Instead of having to start with 0
    - Array assignment
    - Array comparison
    - Array input/output
    - Arrays that know their sizes
    - Arrays that expand dynamically to accommodate more elements

# Example: String Abstract Data Type

- Strings in C++
  - C++ does not provide built-in string data type
    - Maximizes performance
  - Provides mechanisms for creating and implementing string abstract data type
    - String ADT (Chapter 8)
  - ANSI/ISO standard **string** class (Chapter 19)

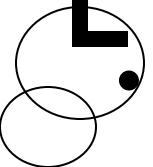
# Example: Queue Abstract Data Type

- Queue
  - FIFO
    - First in, first out
  - Enqueue
    - Put items in queue one at a time
  - Dequeue
    - Remove items from queue one at a time
- Queue ADT
  - Implementation hidden from clients
    - Clients may not manipulate data structure directly
  - Only queue member functions can access internal data
  - Queue ADT (Chapter 15)
  - Standard library **queue** class (Chapter 20)

# Container Classes and Iterators

- Container classes (collection classes)
  - Designed to hold collections of objects
  - Common services
    - Insertion, deletion, searching, sorting, or testing an item
  - Examples
    - Arrays, stacks, queues, trees and linked lists
- Iterator objects (iterators)
  - Returns next item of collection
    - Or performs some action on next item
  - Can have several iterators per container
    - Book with multiple bookmarks
  - Each iterator maintains own “position”
  - Discussed further in Chapter 20

# 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)

```
1 // Fig. 7.20: implementation.h
2 // Header file for class Implementation
3
4 class Implementation {
5
6 public:
7
8     // constructor
9     Implementation( int v )
10    : value( v ) // initialize value with v
11 {
12     // empty body
13
14 } // end Implementation constructor
15
16 // set value to v
17 void setValue( int v )
18 {
19     value = v; // should validate v
20
21 } // end function setValue
22
```

public member function.



## Outline

implementation.h  
(2 of 2)

```
23 // return value
24 int getValue() const
25 {
26     return value;
27 }
28 // end function getValue
29
30 private:
31     int value;
32
33 }; // end class Implementation
```

**public** member function.



## Outline

interface.h (1 of 1)

```

1 // Fig. 7.21: interface.h
2 // Header file for interface.cpp
3
4 class Implementation;           // forward class declaration
5
6 class Interface {
7
8 public:
9     Interface( int );
10    void setValue( int );      // same public interface
11    int getValue() const;     // class Implementation
12    ~Interface();
13
14 private:
15
16    // requires previous forward declaration
17    Implementation *ptr;
18
19 };// 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)

```

1 // Fig. 7.22: interface.cpp
2 // Definition of class Interface
3 #include "interface.h"           // Interface class definition
4 #include "implementation.h"      // Implementation class definition
5
6 // constructor
7 Interface::Interface( int v )   // maintains pointer to underlying
8     : ptr ( new Implementation( v ) ) / Implementation object. Interface
9 {
10    // empty body
11
12 } // end Interface constructor
13
14 // call Implementation's setValue function
15 void Interface::setValue( int v )
16 {
17    ptr->setValue( v );
18
19 } // end function setValue
20

```

Maintain pointer to underlying **Implementation** object.

includes header file for class **Implementation**.

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

## Outline

interface.cpp  
(2 of 2)



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

Deallocate underlying  
**Implementation** object.

```
21 // call Implementation's getValue function
22 int Interface::getValue() const
23 {
24     return ptr->getValue();
25
26 } // end function getValue
27
28 // destructor
29 Interface::~Interface()
30 {
31     delete ptr;
32
33 } // end destructor ~Interface
```



## Outline

fig07\_23.cpp  
(1 of 1)

fig07\_23.cpp  
output (1 of 1)

```

1 // Fig. 7.23: fig07_23.cpp
2 // Hiding a class's private data with a proxy class.
3 #include <iostream>
4
5 using std::cout;
6 using std::endl;
7
8 #include "interface.h" // Interface class definition
9
10 int main()
11 {
12     Interface i( 5 );
13
14     cout << "Interface contains: " << i.getValue()
15     << " before setValue" << endl;
16
17     i.setValue( 10 );
18
19     cout << "Interface contains: " << i.getValue()
20     << " after setValue" << endl;
21
22     return 0;
23
24 } // end main

```

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.

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