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
  – Want to make 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 Bfriend of class A
      – Class A must explicitly declare class Bfriend
  – Not symmetric
    • Class Bfriend of class A
    • Class A not necessarily friend of class B
  – Not transitive
    • Class Afriend of class B
    • Class Bfriend 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
(2 of 3)

```cpp
// 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 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`.

fig07_11.cpp
(3 of 3)

```cpp
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
```
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
// 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;
}

int main()
{
    Test testObject( 12 );
    testObject.print();
    return 0;
}

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

  // 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 );
}

// 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
}

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

// Back to the code...

Return *this as reference to enable cascaded member function calls.
```cpp
// set minute value
Time &Time::setMinute( int m )
{
    minute = ( m >= 0 && m < 60 ) ? m : 0;
    return *this;   // enables cascading
} // end function setMinute

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

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

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

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

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

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

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

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

Function call to **printStandard** must appear last; **printStandard** does not return reference to **t**.
Dynamic Memory Management with Operators

new and delete

- **delete**
  - Destroy dynamically allocated object and free space
  - Consider
    ```cpp
delete timePtr;
```
  - Operator **delete**
    - Calls destructor for object
    - Deallocates memory associated with object
      - Memory can be reused to allocate other objects
  - Deallocating arrays
    ```cpp
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
      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

```
// 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
```

© 2003 Prentice Hall, Inc. All rights reserved.
// 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;
}

// 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;
}

// destructor deallocates dynamically allocated memory
Employee::~Employee()
{
    cout << "~Employee() called for " << firstName
         << ' ' << lastName << endl;
}

// destructor deallocates dynamically allocated memory
Employee::~Employee()
{
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
    • \texttt{int}, \texttt{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

Proxy Classes

• Proxy class
  – Hide implementation details of another class
  – Knows only \texttt{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:

    \begin{verbatim}
    class ClassToLoad;
    \end{verbatim}
// 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
    }
    // set value to v
    void setValue(int v)
    {  // return value
        value = v;  // should validate v
    }
    // return value
    int getValue() const
    {
        return value;
    }
private:
    int value;
};  // end class Implementation

// 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 interface as class Implementation
    int getValue() const;  // call Implementation
    ~Interface();

private:
    // requires previous forward declaration
    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.
// Fig. 7.22: interface.cpp
// Definition of class Interface
#include "interface.h"      // Interface class definition
#include "implementation.h"  // Implementation class definition

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

// call Implementation’s setValue function
void Interface::setValue( int v )
{
    ptr->setValue( v );
}

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

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

Proxy class Interface includes header file for class Implementation.

Maintain pointer to underlying Implementation object.

Invoke corresponding function on underlying Implementation object.

Invoke corresponding function on underlying Implementation object.

Deallocate underlying Implementation object.
// 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