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
      ```cpp
      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

```cpp
// 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.
```cpp
22 // output x
23 void print() const
24 {
25     cout << x << endl;
26 } // end function print
27
28 private:
29     int x; // data member
30 }; // end class Count
31
32 // function setX can modify private data of Count
33 // because setX is a friend of Count
34 void setX( Count &c, int val )
35 {
36     c.x = val; // legal: setX is a friend of Count
37 } // end function setX
38
39 int main()
40 {
41     Count counter;       // create Count object
42     cout << "counter.x after instantiation: ";
43     counter.print();
44     setX( counter, 8 );   // set x with a friend
45     cout << "counter.x after call to setX friend function: ";
46     counter.print();
47     return 0;
48 } // end main
```

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

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

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

```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
```
// 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;
} // end main

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

© 2003 Prentice Hall, Inc. All rights reserved.
# 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

private:
    int hour;    // 0 - 23 (24-hour clock format)
    int minute; // 0 - 59
    int second;  // 0 - 59
}; // end class Time

#endif

Set functions return reference to Time object to enable cascaded member function calls.
// 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 ) ? h : 0;
    return *this;   // enables cascading
} // end function setHour

Return *this as reference to enable cascaded member function calls.
```cpp
43 // set minute value
44 Time &Time::setMinute( int m )
45 {
46     minute = ( m >= 0 && m < 60 ) ? m : 0;
47     return *this; // enables cascading
48 }
49     // end function setMinute
50 // set second value
51     // end function setMinute
52 // set second value
53 Time &Time::setSecond( int s )
54 {
55     second = ( s >= 0 && s < 60 ) ? s : 0;
56     return *this; // enables cascading
57 }
58     // end function setSecond
59 // get hour value
60 int Time::getHour() const
61 {
62     return hour;
63 }
64     // end function getHour
65 // get minute value
66 int Time::getMinute() const
67 {
68     return minute;
69 }
70     // end function getMinute
71 // get second value
72 int Time::getSecond() const
73 {
74     return second;
75 }
76     // end function getSecond
77 // print Time in universal format
78 void Time::printUniversal() const
79 {
80     cout << setfill( '0' ) << setw( 2 ) << hour << ":" << setfill( '0' ) << setw( 2 ) << minute << ":" << setfill( '0' ) << setw( 2 ) << second;
81 }
82 // end function printUniversal
83
84
85
86
87
88
89
```

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

// Fig. 7.16: fig07_16.cpp
// Cascading member function calls with the this pointer.
#include <iostream>
#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: ";
}
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
  
  ```c
  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
    ```c
    double *ptr = new double( 3.14159 );
    Time *timePtr = new Time( 12, 0, 0 );
    ```
  - Allocating arrays
    ```c
    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(::)
      \[ \text{Employee::count} \]
  - private static variables
    - When no class member objects exist: Can only be accessed via
      public static member function
      \[ \text{Employee::getCount()} \]

- 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. 7.18: 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.\n" << endl;
}

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

© 2003 Prentice Hall, Inc.
All rights reserved.
```cpp
48    delete [] firstName; // recapture memory
49    delete [] lastName;  // recapture memory
50
51    --count; // decrement static count of employees
52
53 } // end destructor ~Employee
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
```

---

**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

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;
    ```
// 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 )
{
    value = v; // should validate v
}

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

~Interface();

private:

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

} // end Interface constructor

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

} // end function setValue

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

} // end function getValue

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

} // end destructor ~Interface
```
// 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