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 $C$
// 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.
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
31 }; // end class Count
32
33 // function setX can modify x
34 // because setX is declared
35 // as a friend of Count.
36 void setX( Count &c, int val )
37 {
38    c.x = val;  // legal: setX is a friend of Count
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.
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 \texttt{this} Pointer

\textbf{\texttt{this} pointer}
- Allows object to access own address
- Not part of object itself
  - Implicit argument to non-\texttt{static} member function call
- Implicitly reference member data and functions
- Type of \texttt{this} pointer depends on
  - Type of object
  - Whether member function is \texttt{const}
  - In non-\texttt{const} member function of \texttt{Employee}
    - \texttt{this} has type \texttt{Employee * const}
      - Constant pointer to non-constant \texttt{Employee} object
  - In \texttt{const} member function of \texttt{Employee}
    - \texttt{this} has type \texttt{const Employee * const}
      - Constant pointer to constant \texttt{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
// 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;
51 } // end main

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
    ```cpp
    { 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.
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
// 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
}

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

Return *this as reference to enable cascaded member function calls.
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 Time &Time::setSecond( int s )
52 {
53    second = ( s >= 0 && s < 60 ) ? s : 0;
54    return *this; // enables cascading
55
56 } // end function setSecond

57 // get hour value
58 int Time::getHour() const
59 {
60    return hour;
61
62 } // end function getHour

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

Return *this as reference to enable cascaded member function calls.
// get minute value
int Time::getMinute() const
{
    return minute;
}

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

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


// 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>

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: ";
}
// 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.
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
// 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.
// 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 definitions

// 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.
// 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;
}
```cpp
  delete [] firstName; // recapture memory
  delete [] lastName; // recapture memory
  --count; // decrement static count of employees

  // 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;
  }

  // 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;
  }
```
// 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).
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
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;
    ```
```cpp
// 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
    }

}; // end class Implementation
```

**public** member function.
// return value
int getValue() const
{
    return value;
}
} // end function getValue

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; recall setValue and getValue only public member functions.
    int getValue() const; // class Implementation
    ~Interface();

private:

    // requires previous forward declaration
    Implementation *ptr;

}; // end class Interface
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 )
8     : ptr ( new Implementation( v ) )  // initialize ptr
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
// call Implementation's getValue function
int Interface::getValue() const
{
    return ptr->getValue();
}

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

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