Using Set and Get Functions

• Set functions
  – Perform validity checks before modifying private data
  – Notify if invalid values
  – Indicate with return values
    ```cpp
    void Time::setHour( int h )
    {
      hour = ( h >= 0 && h < 24 ) ? h : 0;
    } // end function setHour
    ```

• Get functions
  – “Query” functions
  – Control format of data returned
    ```cpp
    int Time::getHour()
    {
      return hour;
    } // end function setHour
    ```
Subtle Trap: Returning a Reference to a private Data Member

- **Reference to object**
  - \&pRef = p;
  - Alias for name of object
  - Lvalue
    - Can receive value in assignment statement
      - Changes original object

- **Returning references**
  - **public** member functions can return non-**const** references to **private** data members
    - Client able to modify **private** data members
// Fig. 6.21: time4.h
// Declaration of class Time.
// Member functions defined in time4.cpp

// prevent multiple inclusions of header file
#ifndef TIME4_H
#define TIME4_H

class Time {

public:
    Time( int = 0, int = 0, int = 0 );
    void setTime( int, int, int );
    int getHour();

    int &badSetHour( int );  // DANGEROUS reference return

private:
    int hour;
    int minute;
    int second;

}; // end class Time

#endif

Function to demonstrate effects of returning reference to private data member.
25 // return hour value
26 int Time::getHour()
27 {
28   return hour;
29
30 } // end function getHour
31
32 // POOR PROGRAMMING PRACTICE:
33 // Returning a reference to a private data member.
34 int &Time::badSetHour( int hh )
35 {
36   hour = ( hh >= 0 && hh < 24 ) ? hh : 0;
37
38   return hour; // DANGEROUS reference return
39
40 } // end function badSetHour
// Fig. 6.23: fig06_23.cpp
// Demonstrating a public member function that
// returns a reference to a private data member.
#include <iostream>

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

// include definition of class Time from time4.h
#include "time4.h"

int main()
{
    Time t;

    // store in hourRef the reference returned by badSetHour
    int &hourRef = t.badSetHour( 20 );

    cout << "Hour before modification: " << hourRef;

    // use hourRef to set invalid value in Time object t
    hourRef = 30;

    cout << "\nHour after modification: " << t.getHour();
}
26 // Dangerous: Function call that returns
27 // a reference can be used as an lvalue!
28 t.badSetHour( 12 ) = 74;
29
30 cout << "\n\n************************************************
   " << "POOR PROGRAMMING PRACTICE!!!!!!!
   " << "badSetHour as an lvalue, Hour: " << t.getHour()
31 << "\n************************************************" << endl;
32
33 return 0;
34
35 } // end main

---

Hour before modification: 20
Hour after modification: 30

************************************************
POOR PROGRAMMING PRACTICE!!!!!!!
badSetHour as an lvalue, Hour: 74
************************************************
Default Memberwise Assignment

• Assigning objects
  – Assignment operator (=)
    • Can assign one object to another of same type
    • Default: member-wise assignment
      – Each right member assigned individually to left member

```cpp
class Date {

public:

    Date( int = 1, int = 1, int = 1990 ); // default constructor
    void print();

}; // end class Date
```

• Passing, returning objects
  – Objects passed as function arguments
  – Objects returned from functions
  – Copy constructor
**const (Constant) Objects and const Member Functions**

- **Principle of least privilege**
  - Only allow modification of necessary objects

- **Keyword const**
  - Specify object not modifiable
  - Compiler error if attempt to modify `const` object
  - Example
    ```cpp
    const Time noon( 12, 0, 0 );
    ```
  - Declares `const` object `noon` of class `Time`
  - Initializes to 12
**const (Constant) Objects and const Member Functions**

- **const member functions**
  - Member functions for `const` objects must also be `const`
    - Cannot modify object
  - Specify `const` in both prototype and definition
    - Prototype: After parameter list
      ```
      void printUniversal() const;
      ```
    - Definition: Before beginning left brace
      ```
      void printUniversal() const { .. }  
      ```
- **Constructors and destructors**
  - Cannot be `const`
  - Must be able to modify objects
const (Constant) Objects and const Member Functions

- Member initializer syntax
  - Initializing with member initializer syntax
    - Can be used for
      - All data members
    - Must be used for
      - const data members
      - Data members that are references

- Constructors and destructors
  - Cannot be const
  - Must be able to modify objects
    - Constructor
      - Initializes objects
    - Destructor
      - Performs termination housekeeping
// Fig. 7.4: fig07_04.cpp
// Using a member initializer to initialize a 
// constant of a built-in data type.
#include <iostream>

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

class Increment {
public:
    Increment( int c = 0, int i = 1 ); // default constructor

    void addIncrement();

    void print() const;   // prints count and increment

}; // end class Increment

// function definitions

Increment::Increment( int c = 0, int i = 1 )
{   // default constructor
    count += increment;
}

void Increment::addIncrement()
{   // end function addIncrement
    count += increment;
}

void Increment::print() const   // prints count and increment
{   // end function print
private:
    int count;
    const int increment;  // const data member

}; // end class Increment

// constructor
Increment::Increment( int c, int i ) :
    count( c ),    // initializer for non-const member
    increment( i )  // required initializer for const member
{
    // empty body
}

// print count and increment values
void Increment::print() const
{
    cout << "count = " << count
        << ", increment = " << increment << endl;
}

// Member initializer list
Member initializer syntax can
separated by colon.

// Member initializer syntax can be
must be used for const data

// Member initializer consists of
data member name
would be
(constment) followed by
parentheses containing initial
value (c).
Composition: Objects as Members of Classes

• Composition
  – Class has objects of other classes as members

• Construction of objects
  – Member objects constructed in order declared
    • Not in order of constructor’s member initializer list
    • Constructed before enclosing class objects (host objects)
// Fig. 7.6: date1.h
// Date class definition.
// Member functions defined in date1.cpp
#ifndef DATE1_H
#define DATE1_H

class Date {

public:
    Date( int = 1, int = 1, int = 1900 ); // default constructor
    void print() const; // print date in month/day/year format
    ~Date(); // provided to confirm destruction order

private:
    int month; // 1-12 (January-December)
    int day; // 1-31 based on month
    int year; // any year

    // utility function to test proper day for month and year
    int checkDay( int ) const;

}; // end class Date

#endif
// Fig. 7.7: date1.cpp
// Member-function definitions for class Date.
#include <iostream>

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

// include Date class definition from date1.h
#include "date1.h"

// constructor confirms proper value for month; calls
// utility function checkDay to confirm proper value for day
Date::Date( int mn, int dy, int yr )
{
    if ( mn > 0 && mn <= 12 ) // validate the month
        month = mn;
    else {                     // invalid month set to 1
        month = 1;
        cout << "Month " << mn << " invalid. Set to month 1.\n";
    }

    year = yr;                 // should validate yr
    day = checkDay( dy );      // validate the day
// Fig. 7.8: employee1.h
// Employee class definition.
// Member functions defined in employee1.cpp.
#ifndef EMPLOYEE1_H
#define EMPLOYEE1_H

// include Date class definition from date1.h
#include "date1.h"

class Employee {
public:
    Employee(const char *, const char *, const Date &, const Date &);
    void print() const;
    ~Employee(); // provided to confirm destruction order

private:
    char firstName[ 25 ];
    char lastName[ 25 ];
    const Date birthDate; // composition: member object
    const Date hireDate;   // composition: member object
}; // end class Employee

Using composition;
Employee object contains
Date objects as data members.
// Fig. 7.9: employee1.cpp
// Member-function definitions for class Employee.
#include <iostream>
using std::cout;
using std::endl;
#include <cstring>  // strcpy and strlen prototypes
#include "employee1.h"  // Employee class definition
#include "date1.h"     // Date class definition
// constructor uses member initializer list to pass initializer values to constructors of member objects birthDate and hireDate [Note: This invokes the so-called "default copy constructor" which the C++ compiler provides implicitly.]

Employee::Employee( const char *first, const char *last,
    const Date &dateOfBirth, const Date &dateOfHire )
    : birthDate( dateOfBirth ), // initialize birthDate
      hireDate( dateOfHire )     // initialize hireDate
{
    // copy first into firstName and be sure that it fits
    int length = strlen( first );
    length = ( length < 25 ? length : 24 );
    strncpy( firstName, first, length );
    firstName[ length ] = '\0';

    // copy last into lastName and be sure that it fits
    length = strlen( last );
    length = ( length < 25 ? length : 24 );
    strncpy( lastName, last, length );
    lastName[ length ] = '\0';

    // output Employee object to show when constructor is called
    cout << "Employee object constructor: "
        << firstName << ' ' << lastName << endl;
40 // print Employee object
41 void Employee::print() const
42 {
43     cout << lastName << "", " << firstName << "\nHired: ";
44     hireDate.print();
45     cout << " Birth date: ";
46     birthDate.print();
47     cout << endl;
48 }
49 } // end function print
50
51 // output Employee object to show when it is
52 Employee::~Employee()
53 {
54     cout << "Employee object destructor: "
55     << lastName << "", " << firstName << endl;
56
57 } // end destructor ~Employee

Output to show timing of destructors.
// Fig. 7.10: fig07_10.cpp
// Demonstrating composition—an object with member objects.
#include <iostream>

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

#include "employee1.h" // Employee class definition

int main()
{
    Date birth( 7, 24, 1949 );
    Date hire( 3, 12, 1988 );
    Employee manager( "Bob", "Jones", birth, hire );

    cout << '\n';
    manager.print();

    cout << "\nTest Date constructor with invalid values:\n";
    Date lastDayOff( 14, 35, 1994 );  // invalid month and day
    cout << endl;

    return 0;
} // end main

Create Date objects to pass to Employee constructor.
Date object constructor for date 7/24/1949
Date object constructor for date 3/12/1988
Employee object constructor: Bob Jones

Jones, Bob
Hired: 3/12/1988  Birth date: 7/24/1949

Test Date constructor with invalid values:
Month 14 invalid. Set to month 1.
Day 35 invalid. Set to day 1.
Date object constructor for date 1/1/1994

Date object destructor for date 1/1/1994
Employee object destructor: Jones, Bob
Date object destructor for date 3/12/1988
Date object destructor for date 7/24/1949
Date object destructor for date 3/12/1988
Date object destructor for date 7/24/1949

Note two additional Date objects constructed; no output since default copy constructor used.
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 \textbf{B} \texttt{friend} of class \textbf{A}
      – Class \textbf{A} must explicitly declare class \textbf{B} \texttt{friend}
  – Not symmetric
    • Class \textbf{B} \texttt{friend} of class \textbf{A}
    • Class \textbf{A} not necessarily \texttt{friend} of class \textbf{B}
  – Not transitive
    • Class \textbf{A} \texttt{friend} of class \textbf{B}
    • Class \textbf{B} \texttt{friend} of class \textbf{C}
    • Class \textbf{A} not necessarily \texttt{friend} of Class \textbf{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.
void print() const
{
    cout << x << endl;
}

private:
    int x; // data member
}; // end class Count

// function setX can modify private data of Count
// because setX is declared as a friend of Count
void setX( Count &c, int val )
{
    c.x = val; // legal: setX is a friend of Count
}

Pass Count object since C-style, standalone function.
Since setX friend of Count, can access and modify private data member x.
```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
// 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;

} // end function print

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
    ```
    { 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  }

// set second value
53  Time &Time::setSecond( int s )
54  {
55    second = ( s >= 0 && s < 60 ) ? s : 0;
56    return *this; // enables cascading
57  }

// get hour value
62  int Time::getHour() const
63  {
64    return hour;
65  }

Return *this as reference to enable cascaded member function calls.
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
// 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: ";
}
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

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>

• 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 ];
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 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;
}

// end static function getCount
23  // constructor dynamically allocates space for
24  // first and last name and uses strcpy to copy
25  // first and last names into the object
26  Employee::Employee( const char *first, const char *last )
27  {
28      // dynamically allocates space for first and last name
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 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
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
// 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();
}
```cpp
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 from free-store space
34    cout << "Number of employees after deletion is "
35        << Employee::getCount() << endl;
36
37    return 0;
38
39    } // 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
    • \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 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
    }

}; // end class Implementation
// 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
        int getValue() const;  // class 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 )
{/ * empty body */
    ptr->setValue( v );
} // 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