

Illustrations List [\(Main Page\)](#)

- Fig. 7.1 Using a **Time** class with **const** objects and **const** member functions.
- Fig. 7.2 Using a member initializer to initialize a constant of a built-in data type.
- Fig. 7.3 Erroneous attempt to initialize a constant of a built-in data type by assignment.
- Fig. 7.4 Using member-object initializers.
- Fig. 7.5 Friends can access **private** members of a class.
- Fig. 7.6 Non-**friend**/non-member functions cannot access **private** class members.
- Fig. 7.7 Using the **this** pointer.
- Fig. 7.8 Cascading member function calls.
- Fig. 7.9 Using a **static** data member to maintain a count of the number of objects of a class.
- Fig. 7.10 Implementing a proxy class.

```

1  // Fig. 7.1: time5.h
2  // Declaration of the class Time.
3  // Member functions defined in time5.cpp
4  #ifndef TIME5_H
5  #define TIME5_H
6
7  class Time {
8  public:
9      Time( int = 0, int = 0, int = 0 ); // default constructor
10
11     // set functions
12     void setTime( int, int, int ); // set time
13     void setHour( int ); // set hour
14     void setMinute( int ); // set minute
15     void setSecond( int ); // set second
16
17     // get functions (normally declared const)
18     int getHour() const; // return hour
19     int getMinute() const; // return minute
20     int getSecond() const; // return second
21
22     // print functions (normally declared const)
23     void printMilitary() const; // print military time
24     void printStandard(); // print standard time

```

Fig. 7.1 Using a **Time** class with **const** objects and **const** member functions (part 1 of 6).

```

25 private:
26     int hour; // 0 - 23
27     int minute; // 0 - 59
28     int second; // 0 - 59
29 };
30
31 #endif

```

Fig. 7.1 Using a **Time** class with **const** objects and **const** member functions (part 2 of 6).

```

32 // Fig. 7.1: time5.cpp
33 // Member function definitions for Time class.
34 #include <iostream.h>
35 #include "time5.h"
36
37 // Constructor function to initialize private data.
38 // Default values are 0 (see class definition).
39 Time::Time( int hr, int min, int sec )
40 { setTime( hr, min, sec ); }
41
42 // Set the values of hour, minute, and second.
43 void Time::setTime( int h, int m, int s )
44 {
45     setHour( h );
46     setMinute( m );
47     setSecond( s );
48 }
49
50 // Set the hour value
51 void Time::setHour( int h )
52 { hour = ( h >= 0 && h < 24 ) ? h : 0; }
53
54 // Set the minute value
55 void Time::setMinute( int m )
56 { minute = ( m >= 0 && m < 60 ) ? m : 0; }
57

```

```

58 // Set the second value
59 void Time::setSecond( int s )
60     { second = ( s >= 0 && s < 60 ) ? s : 0; }
61
62 // Get the hour value
63 int Time::getHour() const { return hour; }
64
65 // Get the minute value
66 int Time::getMinute() const { return minute; }
67
68 // Get the second value
69 int Time::getSecond() const { return second; }

```

Fig. 7.1 Using a **Time** class with **const** objects and **const** member functions (part 3 of 6).

```

70
71 // Display military format time: HH:MM
72 void Time::printMilitary() const
73 {
74     cout << ( hour < 10 ? "0" : "" ) << hour << ":"
75         << ( minute < 10 ? "0" : "" ) << minute;
76 }
77
78 // Display standard format time: HH:MM:SS AM (or PM)
79 void Time::printStandard()
80 {
81     cout << ( ( hour == 12 ) ? 12 : hour % 12 ) << ":"
82         << ( minute < 10 ? "0" : "" ) << minute << ":"
83         << ( second < 10 ? "0" : "" ) << second
84         << ( hour < 12 ? " AM" : " PM" );
85 }

```

Fig. 7.1 Using a **Time** class with **const** objects and **const** member functions (part 4 of 6).

```

86 // Fig. 7.1: fig07_01.cpp
87 // Attempting to access a const object with
88 // non-const member functions.
89 #include <iostream.h>
90 #include "time5.h"
91
92 int main()
93 {
94     Time wakeUp( 6, 45, 0 );      // non-constant object
95     const Time noon( 12, 0, 0 );  // constant object
96
97     // MEMBER FUNCTION    OBJECT
98     wakeUp.setHour( 18 );  // non-const    non-const
99
100    noon.setHour( 12 );    // non-const    const
101
102    wakeUp.getHour();      // const        non-const
103
104    noon.getMinute();      // const        const
105    noon.printMilitary();  // const        const
106    noon.printStandard();  // non-const    const
107    return 0;
108 }

```

Fig. 7.1 Using a **Time** class with **const** objects and **const** member functions (part 5 of 6).

```

Compiling Fig07_01.cpp
Fig07_01.cpp(15) : error: 'setHour' :
    cannot convert 'this' pointer from
    'const class Time' to 'class Time &'
    Conversion loses qualifiers
Fig07_01.cpp(21) : error: 'printStandard' :
    cannot convert 'this' pointer from
    'const class Time' to 'class Time &'
    Conversion loses qualifiers

```

Fig. 7.1 Using a **Time** class with **const** objects and **const** member functions (part 6 of 6).

```

1  // Fig. 7.2: fig07_02.cpp
2  // Using a member initializer to initialize a
3  // constant of a built-in data type.
4
5  #include <iostream.h>
6
7  class Increment {
8  public:
9      Increment( int c = 0, int i = 1 );
10     void addIncrement() { count += increment; }
11     void print() const;
12
13 private:
14     int count;
15     const int increment; // const data member
16 };
17
18 // Constructor for class Increment
19 Increment::Increment( int c, int i )
20     : increment( i ) // initializer for const member
21 { count = c; }
22
23 // Print the data
24 void Increment::print() const
25 {
26     cout << "count = " << count
27         << ", increment = " << increment << endl;
28 }
29
30 int main()
31 {
32     Increment value( 10, 5 );
33
34     cout << "Before incrementing: ";
35     value.print();
36
37     for ( int j = 0; j < 3; j++ ) {
38         value.addIncrement();
39         cout << "After increment " << j << ": ";
40         value.print();
41     }
42
43     return 0;
44 }

```

```

Before incrementing: count = 10, increment = 5
After increment 1: count = 15, increment = 5
After increment 2: count = 20, increment = 5
After increment 3: count = 25, increment = 5

```

Fig. 7.2 Using a member initializer to initialize a constant of a built-in data type.

```

1  // Fig. 7.3: fig07_03.cpp
2  // Attempting to initialize a constant of
3  // a built-in data type with an assignment.
4  #include <iostream.h>
5
6  class Increment {
7  public:
8      Increment( int c = 0, int i = 1 );
9      void addIncrement() { count += increment; }
10     void print() const;
11 private:
12     int count;
13     const int increment;
14 };
15
16 // Constructor for class Increment
17 Increment::Increment( int c, int i )
18 {
19     // Constant member 'increment' is not initialized
20     count = c;
21     increment = i; // ERROR: Cannot modify a const object
22 }
23
24 // Print the data
25 void Increment::print() const
26 {
27     cout << "count = " << count
28         << ", increment = " << increment << endl;
29 }
30
31 int main()
32 {
33     Increment value( 10, 5 );
34
35     cout << "Before incrementing: ";
36     value.print();
37
38     for ( int j = 0; j < 3; j++ ) {
39         value.addIncrement();
40         cout << "After increment " << j << ": ";
41         value.print();
42     }
43
44     return 0;
45 }

```

Fig. 7.3 Erroneous attempt to initialize a constant of a built-in data type by assignment (part 1 of 2).

```

Compiling...
Fig7_3.cpp
Fig7_3.cpp(18) : error: 'increment' :
    must be initialized in constructor base/member
    initializer list
Fig7_3.cpp(20) : error: l-value specifies const object

```

Fig. 7.3 Erroneous attempt to initialize a constant of a built-in data type by assignment (part 2 of 2).

```

1  // Fig. 7.4: date1.h
2  // Declaration of the Date class.
3  // Member functions defined in date1.cpp
4  #ifndef DATE1_H
5  #define DATE1_H
6
7  class Date {
8  public:
9      Date( int = 1, int = 1, int = 1900 ); // default constructor
10     void print() const; // print date in month/day/year format
11     ~Date(); // provided to confirm destruction order
12 private:
13     int month; // 1-12
14     int day; // 1-31 based on month
15     int year; // any year
16
17     // utility function to test proper day for month and year
18     int checkDay( int );
19 };
20
21 #endif

```

Fig. 7.4 Using member-object initializers (part 1 of 6).

```

22 // Fig. 7.4: date.cpp
23 // Member function definitions for Date class.
24 #include <iostream.h>
25 #include "date1.h"
26
27 // Constructor: Confirm proper value for month;
28 // call utility function checkDay to confirm proper
29 // value for day.
30 Date::Date( int mn, int dy, int yr )
31 {
32     if ( mn > 0 && mn <= 12 ) // validate the month
33         month = mn;
34     else {
35         month = 1;
36         cout << "Month " << mn << " invalid. Set to month 1.\n";
37     }
38
39     year = yr; // should validate yr
40     day = checkDay( dy ); // validate the day
41
42     cout << "Date object constructor for date ";
43     print(); // interesting: a print with no arguments
44     cout << endl;
45 }
46
47 // Print Date object in form month/day/year

```

```

48 void Date::print() const
49     { cout << month << '/' << day << '/' << year; }
50
51 // Destructor: provided to confirm destruction order
52 Date::~Date()
53 {
54     cout << "Date object destructor for date ";
55     print();
56     cout << endl;
57 }
58

```

Fig. 7.4 Using member-object initializers (part 2 of 6).

```

59 // Utility function to confirm proper day value
60 // based on month and year.
61 // Is the year 2000 a leap year?
62 int Date::checkDay( int testDay )
63 {
64     static const int daysPerMonth[ 13 ] =
65         {0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};
66
67     if ( testDay > 0 && testDay <= daysPerMonth[ month ] )
68         return testDay;
69
70     if ( month == 2 && // February: Check for leap year
71         testDay == 29 &&
72         ( year % 400 == 0 || // year 2000?
73           ( year % 4 == 0 && year % 100 != 0 ) ) ) // year 2000?
74         return testDay;
75
76     cout << "Day " << testDay << " invalid. Set to day 1.\n";
77
78     return 1; // leave object in consistent state if bad value
79 }

```

Fig. 7.4 Using member-object initializers (part 3 of 6).

```

80 // Fig. 7.4: empty1.h
81 // Declaration of the Employee class.
82 // Member functions defined in empty1.cpp
83 #ifndef EMPLOY1_H
84 #define EMPLOY1_H
85
86 #include "date1.h"
87
88 class Employee {
89 public:
90     Employee( char *, char *, int, int, int, int, int, int );
91     void print() const;
92     ~Employee(); // provided to confirm destruction order
93 private:
94     char firstName[ 25 ];
95     char lastName[ 25 ];
96     const Date birthDate;
97     const Date hireDate;
98 };
99
100 #endif

```

Fig. 7.4 Using member-object initializers (part 4 of 6).

```

101 // Fig. 7.4: empty1.cpp
102 // Member function definitions for Employee class.
103 #include <iostream.h>
104 #include <string.h>
105 #include "empty1.h"
106 #include "date1.h"
107
108 Employee::Employee( char *fname, char *lname,
109                    int bmonth, int bday, int byear,
110                    int hmonth, int hday, int hyear )
111     : birthDate( bmonth, bday, byear ),
112       hireDate( hmonth, hday, hyear )
113 {
114     // copy fname into firstName and be sure that it fits
115     int length = strlen( fname );
116     length = ( length < 25 ? length : 24 );
117     strncpy( firstName, fname, length );
118     firstName[ length ] = '\0';
119
120     // copy lname into lastName and be sure that it fits
121     length = strlen( lname );
122     length = ( length < 25 ? length : 24 );
123     strncpy( lastName, lname, length );
124     lastName[ length ] = '\0';
125
126     cout << "Employee object constructor: "
127           << firstName << " " << lastName << endl;
128 }
129
130 void Employee::print() const
131 {
132     cout << lastName << ", " << firstName << "\nHired: ";
133     hireDate.print();
134     cout << " Birth date: ";
135     birthDate.print();
136     cout << endl;
137 }
138
139 // Destructor: provided to confirm destruction order
140 Employee::~Employee()
141 {
142     cout << "Employee object destructor: "
143           << lastName << ", " << firstName << endl;
144 }

```

Fig. 7.4 Using member-object initializers (part 5 of 6).

```

145 // Fig. 7.4: fig07_04.cpp
146 // Demonstrating composition: an object with member objects.
147 #include <iostream.h>
148 #include "empty1.h"
149
150 int main()
151 {
152     Employee e( "Bob", "Jones", 7, 24, 1949, 3, 12, 1988 );
153
154     cout << '\n';
155     e.print();
156
157     cout << "\nTest Date constructor with invalid values:\n";
158     Date d( 14, 35, 1994 ); // invalid Date values
159     cout << endl;
160     return 0;
161 }

```



```

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

```

Fig. 7.4 Using member-object initializers (part 6 of 6).

```

1 // Fig. 7.5: fig07_05.cpp
2 // Friends can access private members of a class.
3 #include <iostream.h>
4
5 // Modified Count class
6 class Count {
7     friend void setX( Count &, int ); // friend declaration
8 public:
9     Count() { x = 0; } // constructor
10    void print() const { cout << x << endl; } // output
11 private:
12    int x; // data member
13 };
14
15 // Can modify private data of Count because
16 // setX is declared as a friend function of Count
17 void setX( Count &c, int val )
18 {
19     c.x = val; // legal: setX is a friend of Count
20 }
21
22 int main()
23 {
24     Count counter;
25
26     cout << "counter.x after instantiation: ";
27     counter.print();
28     cout << "counter.x after call to setX friend function: ";
29     setX( counter, 8 ); // set x with a friend
30     counter.print();
31     return 0;
32 }

```

```

counter.x after instantiation: 0
counter.x after call to setX friend function: 8

```

Fig. 7.5 Friends can access **private** members of a class.

```

1 // Fig. 7.6: fig07_06.cpp
2 // Non-friend/non-member functions cannot access
3 // private data of a class.
4 #include <iostream.h>
5
6 // Modified Count class
7 class Count {
8 public:
9     Count() { x = 0; } // constructor
10    void print() const { cout << x << endl; } // output
11 private:
12    int x; // data member
13 };
14
15 // Function tries to modify private data of Count,
16 // but cannot because it is not a friend of Count.
17 void cannotSetX( Count &c, int val )
18 {
19     c.x = val; // ERROR: 'Count::x' is not accessible
20 }
21
22 int main()
23 {
24     Count counter;
25
26     cannotSetX( counter, 3 ); // cannotSetX is not a friend
27     return 0;
28 }

```

```

Compiling...
Fig07_06.cpp
Fig07_06.cpp(19) : error: 'x' :
    cannot access private member declared in class 'Count'

```

Fig. 7.6 Non-**friend**/non-member functions cannot access **private** class members.

```

1 // Fig. 7.7: fig07_07.cpp
2 // Using the this pointer to refer to object members.
3 #include <iostream.h>
4
5 class Test {
6 public:
7     Test( int = 0 ); // default constructor
8     void print() const;
9 private:
10    int x;
11 };
12
13 Test::Test( int a ) { x = a; } // constructor
14

```

Fig. 7.7 Using the **this** pointer (part 1 of 2).

```

15 void Test::print() const    // ( ) around *this required
16 {
17     cout << "          x = " << x
18         << "\n  this->x = " << this->x
19         << "\n(*this).x = " << ( *this ).x << endl;
20 }
21
22 int main()
23 {
24     Test testObject( 12 );
25
26     testObject.print();
27
28     return 0;
29 }

```

```

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

```

Fig. 7.7 Using the **this** pointer (part 2 of 2).

```

1  // Fig. 7.8: time6.h
2  // Cascading member function calls.
3
4  // Declaration of class Time.
5  // Member functions defined in time6.cpp
6  #ifndef TIME6_H
7  #define TIME6_H
8
9  class Time {
10 public:
11     Time( int = 0, int = 0, int = 0 ); // default constructor
12
13     // set functions
14     Time &setTime( int, int, int ); // set hour, minute, second
15     Time &setHour( int ); // set hour
16     Time &setMinute( int ); // set minute
17     Time &setSecond( int ); // set second
18
19     // get functions (normally declared const)
20     int getHour() const; // return hour
21     int getMinute() const; // return minute
22     int getSecond() const; // return second
23
24     // print functions (normally declared const)
25     void printMilitary() const; // print military time
26     void printStandard() const; // print standard time
27 private:
28     int hour; // 0 - 23
29     int minute; // 0 - 59
30     int second; // 0 - 59
31 };
32
33 #endif

```

Fig. 7.8 Cascading member function calls (part 1 of 4).

```

34 // Fig. 7.8: time.cpp
35 // Member function definitions for Time class.
36 #include "time6.h"
37 #include <iostream.h>
38
39 // Constructor function to initialize private data.
40 // Calls member function setTime to set variables.
41 // Default values are 0 (see class definition).
42 Time::Time( int hr, int min, int sec )
43 { setTime( hr, min, sec ); }
44
45 // Set the values of hour, minute, and second.
46 Time &Time::setTime( int h, int m, int s )
47 {
48     setHour( h );
49     setMinute( m );
50     setSecond( s );
51     return *this;    // enables cascading
52 }
53
54 // Set the hour value
55 Time &Time::setHour( int h )
56 {
57     hour = ( h >= 0 && h < 24 ) ? h : 0;
58
59     return *this;    // enables cascading
60 }
61
62 // Set the minute value
63 Time &Time::setMinute( int m )
64 {
65     minute = ( m >= 0 && m < 60 ) ? m : 0;
66
67     return *this;    // enables cascading
68 }
69
70 // Set the second value
71 Time &Time::setSecond( int s )
72 {
73     second = ( s >= 0 && s < 60 ) ? s : 0;
74
75     return *this;    // enables cascading
76 }
77
78 // Get the hour value
79 int Time::getHour() const { return hour; }
80
81 // Get the minute value
82 int Time::getMinute() const { return minute; }
83

```

Fig. 7.8 Cascading member function calls (part 2 of 4).

```

84 // Get the second value
85 int Time::getSecond() const { return second; }
86
87 // Display military format time: HH:MM
88 void Time::printMilitary() const
89 {
90     cout << ( hour < 10 ? "0" : "" ) << hour << ":"
91         << ( minute < 10 ? "0" : "" ) << minute;
92 }
93
94 // Display standard format time: HH:MM:SS AM (or PM)
95 void Time::printStandard() const
96 {
97     cout << ( ( hour == 0 || hour == 12 ) ? 12 : hour % 12 )
98         << ":" << ( minute < 10 ? "0" : "" ) << minute
99         << ":" << ( second < 10 ? "0" : "" ) << second
100        << ( hour < 12 ? " AM" : " PM" );
101 }

```

Fig. 7.8 Cascading member function calls (part 3 of 4).

```

102 // Fig. 7.8: fig07_08.cpp
103 // Cascading member function calls together
104 // with the this pointer
105 #include <iostream.h>
106 #include "time6.h"
107
108 int main()
109 {
110     Time t;
111
112     t.setHour( 18 ).setMinute( 30 ).setSecond( 22 );
113     cout << "Military time: ";
114     t.printMilitary();
115     cout << "\nStandard time: ";
116     t.printStandard();
117
118     cout << "\n\nNew standard time: ";
119     t.setTime( 20, 20, 20 ).printStandard();
120     cout << endl;
121
122     return 0;
123 }

```

```

Military time: 18:30
Standard time: 6:30:22 PM

New standard time: 8:20:20 PM

```

Fig. 7.8 Cascading member function calls (part 4 of 4).

```

1  // Fig. 7.9: employ1.h
2  // An employee class
3  #ifndef EMPLOY1_H
4  #define EMPLOY1_H
5
6  class Employee {
7  public:
8      Employee( const char*, const char* ); // constructor
9      ~Employee(); // destructor
10     const char *getFirstName() const; // return first name
11     const char *getLastName() const; // return last name
12
13     // static member function
14     static int getCount(); // return # objects instantiated
15
16 private:
17     char *firstName;
18     char *lastName;
19
20     // static data member
21     static int count; // number of objects instantiated
22 };
23
24 #endif

```

Fig. 7.9 Using a **static** data member to maintain a count of the number of objects of a class (part 1 of 5).

```

25 // Fig. 7.9: employ1.cpp
26 // Member function definitions for class Employee
27 #include <iostream.h>
28 #include <string.h>
29 #include <assert.h>
30 #include "employ1.h"
31
32 // Initialize the static data member
33 int Employee::count = 0;
34
35 // Define the static member function that
36 // returns the number of employee objects instantiated.
37 int Employee::getCount() { return count; }
38
39 // Constructor dynamically allocates space for the
40 // first and last name and uses strcpy to copy
41 // the first and last names into the object
42 Employee::Employee( const char *first, const char *last )
43 {
44     firstName = new char[ strlen( first ) + 1 ];
45     assert( firstName != 0 ); // ensure memory allocated
46     strcpy( firstName, first );
47
48     lastName = new char[ strlen( last ) + 1 ];
49     assert( lastName != 0 ); // ensure memory allocated
50     strcpy( lastName, last );
51
52     ++count; // increment static count of employees
53     cout << "Employee constructor for " << firstName
54          << ' ' << lastName << " called." << endl;
55 }
56
57 // Destructor deallocates dynamically allocated memory
58 Employee::~Employee()
59 {
60     cout << "~Employee() called for " << firstName
61          << ' ' << lastName << endl;

```

```

62     delete [] firstName; // recapture memory
63     delete [] lastName;  // recapture memory
64     --count; // decrement static count of employees
65 }
66
67 // Return first name of employee
68 const char *Employee::getFirstName() const
69 {
70     // const before return type prevents client modifying
71     // private data. Client should copy returned string before
72     // destructor deletes storage to prevent undefined pointer.
73     return firstName;
74 }

```

Fig. 7.9 Using a **static** data member to maintain a count of the number of objects of a class (part 2 of 5).

```

75
76 // Return last name of employee
77 const char *Employee::getLastName() const
78 {
79     // const before return type prevents client modifying
80     // private data. Client should copy returned string before
81     // destructor deletes storage to prevent undefined pointer.
82     return lastName;
83 }

```

Fig. 7.9 Using a **static** data member to maintain a count of the number of objects of a class (part 3 of 5).

```

84 // Fig. 7.9: fig07_09.cpp
85 // Driver to test the Employee class
86 #include <iostream.h>
87 #include "employ1.h"
88
89 int main()
90 {
91     cout << "Number of employees before instantiation is "
92         << Employee::getCount() << endl; // use class name
93
94     Employee *e1Ptr = new Employee( "Susan", "Baker" );
95     Employee *e2Ptr = new Employee( "Robert", "Jones" );
96
97     cout << "Number of employees after instantiation is "
98         << e1Ptr->getCount();
99
100    cout << "\n\nEmployee 1: "
101        << e1Ptr->getFirstName()
102        << " " << e1Ptr->getLastName()
103        << "\nEmployee 2: "
104        << e2Ptr->getFirstName()
105        << " " << e2Ptr->getLastName() << "\n\n";
106
107    delete e1Ptr; // recapture memory
108    e1Ptr = 0;
109    delete e2Ptr; // recapture memory
110    e2Ptr = 0;
111
112    cout << "Number of employees after deletion is "
113        << Employee::getCount() << endl;
114
115    return 0;
116 }

```

Fig. 7.9 Using a **static** data member to maintain a count of the number of objects of a class (part 4 of 5).

```

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

```

Fig. 7.9 Using a **static** data member to maintain a count of the number of objects of a class (part 5 of 5).

```

1 // Fig. 7.10: implementation.h
2 // Header file for class Implementation
3
4 class Implementation {
5     public:
6         Implementation( int v ) { value = v; }
7         void setValue( int v ) { value = v; }
8         int getValue() const { return value; }
9
10    private:
11        int value;
12 };

```

Fig. 7.10 Implementing a proxy class (part 1 of 4).

```

13 // Fig. 7.10: interface.h
14 // Header file for interface.cpp
15 class Implementation; // forward class declaration
16
17 class Interface {
18     public:
19         Interface( int );
20         void setValue( int ); // same public interface as
21         int getValue() const; // class Implementation
22     private:
23         Implementation *ptr; // requires previous
24                               // forward declaration
25 };

```

Fig. 7.10 Implementing a proxy class (part 2 of 4).


```

26 // Fig. 7.10: interface.cpp
27 // Definition of class Interface
28 #include "interface.h"
29 #include "implementation.h"
30
31 Interface::Interface( int v )
32     : ptr ( new Implementation( v ) ) { }
33
34 // call Implementation's setValue function
35 void Interface::setValue( int v ) { ptr->setValue( v ); }
36
37 // call Implementation's getValue function
38 int Interface::getValue() const { return ptr->getValue(); }

```

Fig. 7.10 Implementing a proxy class (part 3 of 4).

```

39 // Fig. 7.10: fig07_10.cpp
40 // Hiding a class's private data with a proxy class.
41 #include <iostream.h>
42 #include "interface.h"
43
44 int main()
45 {
46     Interface i( 5 );
47
48     cout << "Interface contains: " << i.getValue()
49         << " before setValue" << endl;
50     i.setValue( 10 );
51     cout << "Interface contains: " << i.getValue()
52         << " after setValue" << endl;
53     return 0;
54 }

```

```

Interface contains: 5 before setVal
Interface contains: 10 after setVal

```

Fig. 7.10 Implementing a proxy class (part 4 of 4).