

# 10

## Classes: A Deeper Look, Part 2



*But what, to serve our private ends, Forbids the cheating of our friends?*

— Charles Churchill

Instead of this absurd division into sexes they ought to class people as static and dynamic.

— Evelyn Waugh

Have no friends not equal to yourself.

— Confucius



# OBJECTIVES

In this chapter you will learn:

- To specify `const` (constant) objects and `const` member functions.
- To create objects composed of other objects.
- To use `friend` functions and `friend` classes.
- To use the `this` pointer.
- To create and destroy objects dynamically with operators `new` and `delete`, respectively.
- To use `static` data members and member functions.
- The concept of a container class.
- The notion of iterator classes that walk through the elements of container classes.
- To use proxy classes to hide implementation details from a class's clients.



- 10.1** Introduction
- 10.2** `const` (Constant) Objects and `const` Member Functions
- 10.3** Composition: Objects as Members of Classes
- 10.4** `friend` Functions and `friend` Classes
- 10.5** Using the `this` Pointer
- 10.6** Dynamic Memory Management with Operators `new` and `delete`
- 10.7** `static` Class Members
- 10.8** Data Abstraction and Information Hiding
  - 10.8.1** Example: Array Abstract Data Type
  - 10.8.2** Example: String Abstract Data Type
  - 10.8.3** Example: Queue Abstract Data Type
- 10.9** Container Classes and Iterators
- 10.10** Proxy Classes
- 10.11** Wrap-Up



# 10.1 Introduction

- **const objects and const member functions**
  - Prevent modifications of objects
  - Enforce the principle of least privilege
- **Composition**
  - Classes having objects of other classes as members
- **Friendship**
  - Enables class designer to specify that certain non-member functions can access the class's non-**public** members



## 10.1 Introduction (Cont.)

- **this** pointer
- **Dynamic memory management**
  - **new** and **delete** operators
- **static** class members
- **Proxy classes**
  - Hide implementation details of a class from clients
- **Pointer-base strings**
  - Used in C legacy code from the last two decades



# 10.2 `const` (Constant) Objects and `const` Member Functions

- **Principle of least privilege**
  - One of the most fundamental principles of good software engineering
  - Applies to objects, too
- **`const` objects**
  - Keyword `const`
  - Specifies that an object is not modifiable
  - Attempts to modify the object will result in compilation errors



# Software Engineering Observation 10.1

---

**Declaring an object as `const` helps enforce the principle of least privilege. Attempts to modify the object are caught at compile time rather than causing execution-time errors. Using `const` properly is crucial to proper class design, program design and coding.**



## Performance Tip 10.1

---

**Declaring variables and objects `const` can improve performance—today’s sophisticated optimizing compilers can perform certain optimizations on constants that cannot be performed on variables.**



## 10.2 `const` (Constant) Objects and `const` Member Functions (Cont.)

- **`const` member functions**

- Only **`const`** member function can be called for **`const`** objects
- Member functions declared **`const`** are not allowed to modify the object
- A function is specified as **`const`** both in its prototype and in its definition
- **`const`** declarations are not allowed for constructors and destructors



# Common Programming Error 10.1

---

**Defining as `const` a member function that modifies a data member of an object is a compilation error.**



## Common Programming Error 10.2

---

**Defining as `const` a member function that calls a non-`const` member function of the class on the same instance of the class is a compilation error.**



## Common Programming Error 10.3

---

**Invoking a non-`const` member function on a `const` object is a compilation error.**



## Software Engineering Observation 10.2

---

**A `const` member function can be overloaded with a non-`const` version. The compiler chooses which overloaded member function to use based on the object on which the function is invoked. If the object is `const`, the compiler uses the `const` version. If the object is not `const`, the compiler uses the non-`const` version.**



## Common Programming Error 10.4

---

**Attempting to declare a constructor or destructor `const` is a compilation error.**



```
1 // Fig. 10.1: Time.h
2 // Definition of class Time.
3 // Member functions defined in Time.cpp.
4 #ifndef TIME_H
5 #define TIME_H
6
7 class Time
8 {
9 public:
10     Time( int = 0, int = 0, int = 0 ); // default constructor
11
12     // set functions
13     void setTime( int, int, int ); // set time
14     void setHour( int ); // set hour
15     void setMinute( int ); // set minute
16     void setSecond( int ); // set second
17
18     // get functions (normally declared const)
19     int getHour() const; // return hour
20     int getMinute() const; // return minute
21     int getSecond() const; // return second
```

**const** keyword to indicate that member function cannot modify the object



## Outline

Time.h

(2 of 2)

```
22 // print functions (normally declared const)
23 void printUniversal() const; // print universal time
24 void printStandard(); // print standard time (should be const)
25 private:
26 int hour; // 0 - 23 (24-hour clock format)
27 int minute; // 0 - 59
28 int second; // 0 - 59
29 }; // end class Time
30
31
32 #endif
```



## Outline

Time.cpp

(1 of 3)

```
1 // Fig. 10.2: Time.cpp
2 // Member-function definitions for class Time.
3 #include <iostream>
4 using std::cout;
5
6 #include <iomanip>
7 using std::setfill;
8 using std::setw;
9
10 #include "Time.h" // include definition of class Time
11
12 // constructor function to initialize private data;
13 // calls member function setTime to set variables;
14 // default values are 0 (see class definition)
15 Time::Time( int hour, int minute, int second )
16 {
17     setTime( hour, minute, second );
18 } // end Time constructor
19
20 // set hour, minute and second values
21 void Time::setTime( int hour, int minute, int second )
22 {
23     setHour( hour );
24     setMinute( minute );
25     setSecond( second );
26 } // end function setTime
```



## Outline

Time.cpp

(2 of 3)

```
27
28 // set hour value
29 void Time::setHour( int h )
30 {
31     hour = ( h >= 0 && h < 24 ) ? h : 0; // validate hour
32 } // end function setHour
33
34 // set minute value
35 void Time::setMinute( int m )
36 {
37     minute = ( m >= 0 && m < 60 ) ? m : 0; // validate minute
38 } // end function setMinute
39
40 // set second value
41 void Time::setSecond( int s )
42 {
43     second = ( s >= 0 && s < 60 ) ? s : 0; // validate second
44 } // end function setSecond
45
46 // return hour value
47 int Time::getHour() const // get functions should be const
48 {
49     return hour;
50 } // end function getHour
```

const keyword in function definition, as well as in function prototype



## Outline

Time.cpp

(3 of 3)

```
51
52 // return minute value
53 int Time::getMinute() const
54 {
55     return minute;
56 } // end function getMinute
57
58 // return second value
59 int Time::getSecond() const
60 {
61     return second;
62 } // end function getSecond
63
64 // print Time in universal-time format (HH:MM:SS)
65 void Time::printUniversal() const
66 {
67     cout << setfill( '0' ) << setw( 2 ) << hour << ":"
68         << setw( 2 ) << minute << ":" << setw( 2 ) << second;
69 } // end function printUniversal
70
71 // print Time in standard-time format (HH:MM:SS AM or PM)
72 void Time::printStandard() // note lack of const declaration
73 {
74     cout << ( ( hour == 0 || hour == 12 ) ? 12 : hour % 12 )
75         << ":" << setfill( '0' ) << setw( 2 ) << minute
76         << ":" << setw( 2 ) << second << ( hour < 12 ? " AM" : " PM" );
77 } // end function printStandard
```



## Outline

fig10\_03.cpp

(1 of 2)

```

1 // Fig. 10.3: fig10_03.cpp
2 // Attempting to access a const object with non-const member functions.
3 #include "Time.h" // include Time class definition
4
5 int main()
6 {
7     Time wakeUp( 6, 45, 0 ); // non-constant object
8     const Time noon( 12, 0, 0 ); // constant object
9
10         // OBJECT      MEMBER FUNCTION
11     wakeUp.setHour( 18 ); // non-const non-const
12
13     noon.setHour( 12 ); // const non-const
14
15     wakeUp.getHour(); // non-const const
16
17     noon.getMinute(); // const const
18     noon.printUniversal(); // const const
19
20     noon.printStandard(); // const non-const
21     return 0;
22 } // end main

```

Cannot invoke non-**const** member functions on a **const** object



## Outline

fig10\_03.cpp

(2 of 2)

*Borland C++ command-line compiler error messages:*

```
Warning W8037 fig10_03.cpp 13: Non-const function Time::setHour(int)
    called for const object in function main()
Warning W8037 fig10_03.cpp 20: Non-const function Time::printStandard()
    called for const object in function main()
```

*Microsoft Visual C++.NET compiler error messages:*

```
C:\cpphttp5_examples\ch10\Fig10_01_03\fig10_03.cpp(13) : error C2662:
'Time::setHour' : cannot convert 'this' pointer from 'const Time' to
'Time &'
    Conversion loses qualifiers
C:\cpphttp5_examples\ch10\Fig10_01_03\fig10_03.cpp(20) : error C2662:
'Time::printStandard' : cannot convert 'this' pointer from 'const Time' to
'Time &'
    Conversion loses qualifiers
```

*GNU C++ compiler error messages:*

```
fig10_03.cpp:13: error: passing `const Time' as `this' argument of
`void Time::setHour(int)' discards qualifiers
fig10_03.cpp:20: error: passing `const Time' as `this' argument of
`void Time::printStandard()' discards qualifiers
```



# 10.2 `const` (Constant) Objects and `const` Member Functions (Cont.)

- **Member initializer**
  - Required for initializing
    - `const` data members
    - Data members that are references
  - Can be used for any data member
- **Member initializer list**
  - Appears between a constructor's parameter list and the left brace that begins the constructor's body
  - Separated from the parameter list with a colon (`:`)
  - Each member initializer consists of the data member name followed by parentheses containing the member's initial value
  - Multiple member initializers are separated by commas
  - Executes before the body of the constructor executes



## Outline

Increment.h

(1 of 1)

```
1 // Fig. 10.4: Increment.h
2 // Definition of class Increment.
3 #ifndef INCREMENT_H
4 #define INCREMENT_H
5
6 class Increment
7 {
8 public:
9     Increment( int c = 0, int i = 1 ); // default constructor
10
11     // function addIncrement definition
12     void addIncrement()
13     {
14         count += increment;
15     } // end function addIncrement
16
17     void print() const; // prints count and increment
18 private:
19     int count;
20     const int increment; // const data member
21 }; // end class Increment
22
23 #endif
```

**const** data member that must be initialized using a member initializer



## Outline

Increment.cpp

(1 of 1)

```

1 // Fig. 10.5: Increment.cpp
2 // Member-function definitions for class Increment demonstrate using a
3 // member initializer to initialize a constant of a built-in data type.
4 #include <iostream>
5 using std::cout;
6 using std::endl;
7
8 #include "Increment.h" // increment definitions for Increment
9
10 // constructor
11 Increment::Increment( int c, int i )
12     : count( c ), // initializer for non-const member
13       increment( i ) // required initializer for const member
14 {
15     // empty body
16 } // end constructor Increment
17
18 // print count and increment values
19 void Increment::print() const
20 {
21     cout << "count = " << count << ", increment = " << increment << endl;
22 } // end function print

```

Colon (:) marks the start of a member initializer list

Member initializer for non-const member **count**

Required member initializer for **const** member **increment**

## Outline

fig10\_06.cpp

(1 of 1)

```
1 // Fig. 10.6: fig10_06.cpp
2 // Program to test class Increment.
3 #include <iostream>
4 using std::cout;
5
6 #include "Increment.h" // include definition of class Increment
7
8 int main()
9 {
10     Increment value( 10, 5 );
11
12     cout << "Before incrementing: ";
13     value.print();
14
15     for ( int j = 1; j <= 3; j++ )
16     {
17         value.addIncrement();
18         cout << "After increment " << j << ": ";
19         value.print();
20     } // end for
21
22     return 0;
23 } // end main
```

```
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
```



## Software Engineering Observation 10.3

---

**A `const` object cannot be modified by assignment, so it must be initialized. When a data member of a class is declared `const`, a member initializer must be used to provide the constructor with the initial value of the data member for an object of the class. The same is true for references.**



# Common Programming Error 10.5

---

**Not providing a member initializer for a `const` data member is a compilation error.**



## Software Engineering Observation 10.4

---

**Constant data members (`const` objects and `const` variables) and data members declared as references must be initialized with member initializer syntax; assignments for these types of data in the constructor body are not allowed.**



## Error-Prevention Tip 10.1

---

Declare as **const** all of a class's member functions that do not modify the object in which they operate. Occasionally this may seem inappropriate, because you will have no intention of creating **const** objects of that class or accessing objects of that class through **const** references or pointers to **const**. Declaring such member functions **const** does offer a benefit, though. If the member function is inadvertently written to modify the object, the compiler will issue an error message.

---



```
1 // Fig. 10.7: Increment.h
2 // Definition of class Increment.
3 #ifndef INCREMENT_H
4 #define INCREMENT_H
5
6 class Increment
7 {
8 public:
9     Increment( int c = 0, int i = 1 ); // default constructor
10
11     // function addIncrement definition
12     void addIncrement()
13     {
14         count += increment;
15     } // end function addIncrement
16
17     void print() const; // prints count and increment
18 private:
19     int count;
20     const int increment; // const data member
21 }; // end class Increment
22
23 #endif
```

Member function declared **const** to prevent errors in situations where an **Increment** object is treated as a **const** object



## Outline

Increment.cpp

(1 of 1)

```
1 // Fig. 10.8: Increment.cpp
2 // Attempting to initialize a constant of
3 // a built-in data type with an assignment.
4 #include <iostream>
5 using std::cout;
6 using std::endl;
7
8 #include "Increment.h" // include definition of class Increment
9
10 // constructor; constant member 'increment' is not initialized
11 Increment::Increment( int c, int i )
12 {
13     count = c; // allowed because count is not constant
14     increment = i; // ERROR: Cannot modify a const object
15 } // end constructor Increment
16
17 // print count and increment values
18 void Increment::print() const
19 {
20     cout << "count = " << count << ", increment = " << increment << endl;
21 } // end function print
```

It is an error to modify a **const** data member; data member **increment** must be initialized with a member initializer



## Outline

fig10\_09.cpp

(1 of 2)

```
1 // Fig. 10.9: fig10_09.cpp
2 // Program to test class Increment.
3 #include <iostream>
4 using std::cout;
5
6 #include "Increment.h" // include definition of class Increment
7
8 int main()
9 {
10     Increment value( 10, 5 );
11
12     cout << "Before incrementing: ";
13     value.print();
14
15     for ( int j = 1; j <= 3; j++ )
16     {
17         value.addIncrement();
18         cout << "After increment " << j << ": ";
19         value.print();
20     } // end for
21
22     return 0;
23 } // end main
```



## Outline

fig10\_09.cpp

(2 of 2)

*Borland C++ command-line compiler error message:*

```
Error E2024 Increment.cpp 14: Cannot modify a const object in function
Increment::Increment(int,int)
```

*Microsoft Visual C++.NET compiler error messages:*

```
C:\cpphttp5_examples\ch10\Fig10_07_09\Increment.cpp(12) : error C2758:
'Increment::increment' : must be initialized in constructor
base/member initializer list
    C:\cpphttp5_examples\ch10\Fig10_07_09\Increment.h(20) :
        see declaration of 'Increment::increment'
C:\cpphttp5_examples\ch10\Fig10_07_09\Increment.cpp(14) : error C2166:
l-value specifies const object
```

*GNU C++ compiler error messages:*

```
Increment.cpp:12: error: uninitialized member 'Increment::increment' with
'const' type 'const int'
Increment.cpp:14: error: assignment of read-only data-member
'Increment::increment'
```



# 10.3 Composition: Objects as Members of Classes

- **Composition**

- Sometimes referred to as a *has-a* relationship
- A class can have objects of other classes as members
- Example
  - **AlarmClock** object with a **Time** object as a member



## 10.3 Composition: Objects as Members of Classes (Cont.)

- **Initializing member objects**
  - **Member initializers pass arguments from the object's constructor to member-object constructors**
  - **Member objects are constructed in the order in which they are declared in the class definition**
    - **Not in the order they are listed in the constructor's member initializer list**
    - **Before the enclosing class object (host object) is constructed**
  - **If a member initializer is not provided**
    - **The member object's default constructor will be called implicitly**



# Software Engineering Observation 10.5

---

**A common form of software reusability is composition, in which a class has objects of other classes as members.**



## Outline

Date.h

(1 of 1)

```
1 // Fig. 10.10: Date.h
2 // Date class definition; Member functions defined in Date.cpp
3 #ifndef DATE_H
4 #define DATE_H
5
6 class Date
7 {
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 (January-December)
14    int day; // 1-31 based on month
15    int year; // any year
16
17    // utility function to check if day is proper for month and year
18    int checkDay( int ) const;
19 }; // end class Date
20
21 #endif
```



## Outline

Date.cpp

(1 of 3)

```
1 // Fig. 10.11: Date.cpp
2 // Member-function definitions for class Date.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 #include "Date.h" // include Date class definition
8
9 // constructor confirms proper value for month; calls
10 // utility function checkDay to confirm proper value for day
11 Date::Date( int mn, int dy, int yr )
12 {
13     if ( mn > 0 && mn <= 12 ) // validate the month
14         month = mn;
15     else
16     {
17         month = 1; // invalid month set to 1
18         cout << "Invalid month (" << mn << ") set to 1.\n";
19     } // end else
20
21     year = yr; // could validate yr
22     day = checkDay( dy ); // validate the day
23
24     // output Date object to show when its constructor is called
25     cout << "Date object constructor for date ";
26     print();
27     cout << endl;
28 } // end Date constructor
```



## Outline

Date.cpp

(2 of 3)

```
29
30 // print Date object in form month/day/year
31 void Date::print() const
32 {
33     cout << month << '/' << day << '/' << year;
34 } // end function print
35
36 // output Date object to show when its destructor is called
37 Date::~Date()
38 {
39     cout << "Date object destructor for date ";
40     print();
41     cout << endl;
42 } // end ~Date destructor
```



## Outline

Date.cpp

(3 of 3)

```
43
44 // utility function to confirm proper day value based on
45 // month and year; handles leap years, too
46 int Date::checkDay( int testDay ) const
47 {
48     static const int daysPerMonth[ 13 ] =
49         { 0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 };
50
51     // determine whether testDay is valid for specified month
52     if ( testDay > 0 && testDay <= daysPerMonth[ month ] )
53         return testDay;
54
55     // February 29 check for leap year
56     if ( month == 2 && testDay == 29 && ( year % 400 == 0 ||
57         ( year % 4 == 0 && year % 100 != 0 ) ) )
58         return testDay;
59
60     cout << "Invalid day (" << testDay << ") set to 1.\n";
61     return 1; // leave object in consistent state if bad value
62 } // end function checkDay
```



```
1 // Fig. 10.12: Employee.h
2 // Employee class definition.
3 // Member functions defined in Employee.cpp.
4 #ifndef EMPLOYEE_H
5 #define EMPLOYEE_H
6
7 #include "Date.h" // include Date class definition
8
9 class Employee
10 {
11 public:
12     Employee( const char * const, const char * const,
13             const Date &, const Date & );
14     void print() const;
15     ~Employee(); // provided to confirm deconstruction
16 private:
17     char firstName[ 25 ];
18     char lastName[ 25 ];
19     const Date birthDate; // composition: member object
20     const Date hireDate; // composition: member object
21 }; // end class Employee
22
23 #endif
```

Parameters to be passed via member  
initializers to the constructor for class **Date**

**const** objects of class **Date** as members



## Outline

Employee.cpp

(1 of 2)

```
1 // Fig. 10.13: Employee.cpp
2 // Member-function definitions for class Employee.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 #include <cstring> // strlen and strncpy prototypes
8 using std::strlen;
9 using std::strncpy;
10
11 #include "Employee.h" // Employee class definition
12 #include "Date.h" // Date class definition
13
14 // constructor uses member initializer list to pass initializer
15 // values to constructors of member objects birthDate and hireDate
16 // [Note: This invokes the so-called "default copy constructor" which the
17 // C++ compiler provides implicitly.]
18 Employee::Employee( const char * const first, const char * const last,
19     const Date &dateOfBirth, const Date &dateOfHire )
20     : birthDate( dateOfBirth ) // initialize birthDate
21     , hireDate( dateOfHire ) // initialize hireDate
22 {
23     // copy first into firstName and be
24     int length = strlen( first );
25     length = ( length < 25 ? length : 24 );
26     strncpy( firstName, first, length );
27     firstName[ length ] = '\0';
```

Member initializers that pass arguments to **Date**'s implicit default copy constructor



## Outline

Employee.cpp

(2 of 2)

```
28 // copy last into lastName and be sure that it fits
29 length = strlen( last );
30 length = ( length < 25 ? length : 24 );
31 strncpy( lastName, last, length );
32 lastName[ length ] = '\0';
33
34
35 // output Employee object to show when constructor is called
36 cout << "Employee object constructor: "
37     << firstName << " " << lastName << endl;
38 } // end Employee constructor
39
40 // print Employee object
41 void Employee::print() const
42 {
43     cout << lastName << ", " << firstName << " Hired: ";
44     hireDate.print();
45     cout << " Birthday: ";
46     birthDate.print();
47     cout << endl;
48 } // end function print
49
50 // output Employee object to show when its destructor is called
51 Employee::~Employee()
52 {
53     cout << "Employee object destructor: "
54         << lastName << ", " << firstName << endl;
55 } // end ~Employee destructor
```



## Outline

fig10\_14.cpp

(1 of 2)

```
1 // Fig. 10.14: fig10_14.cpp
2 // Demonstrating composition--an object with member objects.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 #include "Employee.h" // Employee class definition
8
9 int main()
10 {
11     Date birth( 7, 24, 1949 );
12     Date hire( 3, 12, 1988 );
13     Employee manager( "Bob", "Blue", birth, hire );
14
15     cout << endl;
16     manager.print();
17
18     cout << "\nTest Date constructor with invalid values:\n";
19     Date lastDayOff( 14, 35, 1994 ); // invalid month and day
20     cout << endl;
21     return 0;
22 } // end main
```



Passing objects to a host object constructor



# Outline

fig10\_14.cpp

(2 of 2)

```
Date object constructor for date 7/24/1949
Date object constructor for date 3/12/1988
Employee object constructor: Bob Blue
```

```
Blue, Bob Hired: 3/12/1988 Birthday: 7/24/1949
```

```
Test Date constructor with invalid values:
Invalid month (14) set to 1.
Invalid day (35) set to 1.
Date object constructor for date 1/1/1994
```

```
Date object destructor for date 1/1/1994
Employee object destructor: Blue, 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
```



## Common Programming Error 10.6

---

**A compilation error occurs if a member object is not initialized with a member initializer and the member object's class does not provide a default constructor (i.e., the member object's class defines one or more constructors, but none is a default constructor).**



## Performance Tip 10.2

---

**Initialize member objects explicitly through member initializers. This eliminates the overhead of “doubly initializing” member objects—once when the member object’s default constructor is called and again when *set* functions are called in the constructor body (or later) to initialize the member object.**



## Software Engineering Observation 10.6

---

**If a class member is an object of another class, making that member object `public` does not violate the encapsulation and hiding of that member object's `private` members. However, it does violate the encapsulation and hiding of the containing class's implementation, so member objects of class types should still be `private`, like all other data members.**



# 10.4 friend Functions and friend Classes

- **friend function of a class**
  - **Defined outside that class's scope**
    - **Not a member function of that class**
  - **Yet has the right to access the non-public (and public) members of that class**
  - **Standalone functions or entire classes may be declared to be friends of a class**
  - **Can enhance performance**
  - **Often appropriate when a member function cannot be used for certain operations**



## 10.4 friend Functions and friend Classes (Cont.)

- **To declare a function as a friend of a class:**
  - Provide the function prototype in the class definition preceded by keyword **friend**
- **To declare a class as a friend of a class:**
  - Place a declaration of the form  
**friend class ClassTwo;**  
in the definition of class **ClassOne**
    - All member functions of class **ClassTwo** are **friends** of class **ClassOne**



## 10.4 friend Functions and friend Classes (Cont.)

- **Friendship is granted, not taken**
  - For class **B** to be a friend of class **A**, class **A** must explicitly declare that class **B** is its friend
- **Friendship relation is neither symmetric nor transitive**
  - If class **A** is a friend of class **B**, and class **B** is a friend of class **C**, you cannot infer that class **B** is a friend of class **A**, that class **C** is a friend of class **B**, or that class **A** is a friend of class **C**
- **It is possible to specify overloaded functions as friends of a class**
  - Each overloaded function intended to be a **friend** must be explicitly declared as a **friend** of the class



# Software Engineering Observation 10.7

---

**Even though the prototypes for `friend` functions appear in the class definition, friends are not member functions.**



# Software Engineering Observation 10.8

---

**Member access notions of `private`, `protected` and `public` are not relevant to `friend` declarations, so `friend` declarations can be placed anywhere in a class definition.**



## Good Programming Practice 10.1

---

**Place all friendship declarations first inside the class definition's body and do not precede them with any access specifier.**



# Software Engineering Observation 10.9

---

**Some people in the OOP community feel that “friendship” corrupts information hiding and weakens the value of the object-oriented design approach. In this text, we identify several examples of the responsible use of friendship.**



## Outline

fig10\_15.cpp

(1 of 2)

```
1 // Fig. 10.15: fig10_15.cpp
2 // Friends can access private members of a class.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 // Count class definition
8 class Count
9 {
10     friend void setX( Count &, int ); // friend declaration
11 public:
12     // constructor
13     Count()
14         : x( 0 ) // initialize x to 0
15     {
16         // empty body
17     } // end constructor Count
18
19     // output x
20     void print() const
21     {
22         cout << x << endl;
23     } // end function print
24 private:
25     int x; // data member
26 }; // end class Count
```

friend function declaration (can appear anywhere in the class)



```
27
28 // function setX can modify private data of Count
29 // because setX is declared as a friend of Count (line 10)
30 void setX( Count &c, int val )
31 {
32     c.x = val; // allowed because setX is a friend of Count
33 } // end function setX
34
35 int main()
36 {
37     Count counter; // create Count object
38
39     cout << "counter.x after instantiation: ";
40     counter.print();
41
42     setX( counter, 8 ); // set x using a friend function
43     cout << "counter.x after call to setX friend function: ";
44     counter.print();
45     return 0;
46 } // end main
```

fig10\_15.cpp

friend function can modify Count's private data

Calling a friend function; note that we pass the Count object to the function

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



## Outline

fig10\_16.cpp

(1 of 3)

```
1 // Fig. 10.16: fig10_16.cpp
2 // Non-friend/non-member functions cannot access private data of a class.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 // Count class definition (note that there is no friendship declaration)
8 class Count
9 {
10 public:
11     // constructor
12     Count()
13         : x( 0 ) // initialize x to 0
14     {
15         // empty body
16     } // end constructor Count
17
18     // output x
19     void print() const
20     {
21         cout << x << endl;
22     } // end function print
23 private:
24     int x; // data member
25 }; // end class Count
```



Non-**friend** function cannot access the class's **private** data

```
26 // function cannotSetX tries to modify private data of Count,
27 // but cannot because the function is not a friend of Count
28 void cannotSetX( Count &c, int val )
29 {
30     c.x = val; // ERROR: cannot access private member in Count
31 } // end function cannotSetX
32
33
34 int main()
35 {
36     Count counter; // create Count object
37
38     cannotSetX( counter, 3 ); // cannotSetX is not a friend
39     return 0;
40 } // end main
```

fig10\_16.cpp

(2 of 3)



## Outline

fig10\_16.cpp

(3 of 3)

*Borland C++ command-line compiler error message:*

```
Error E2247 Fig10_16/fig10_16.cpp 31: 'Count::x' is not accessible in
function cannotSetX(Count &,int)
```

*Microsoft Visual C++.NET compiler error messages:*

```
C:\cpphttp5_examples\ch10\Fig10_16\fig10_16.cpp(31) : error C2248: 'Count::x'
: cannot access private member declared in class 'Count'
    C:\cpphttp5_examples\ch10\Fig10_16\fig10_16.cpp(24) : see declaration
of 'Count::x'
    C:\cpphttp5_examples\ch10\Fig10_16\fig10_16.cpp(9) : see declaration
of 'Count'
```

*GNU C++ compiler error messages:*

```
fig10_16.cpp:24: error: 'int Count::x' is private
fig10_16.cpp:31: error: within this context
```



# 10.5 Using the `this` Pointer

- **Member functions know which object's data members to manipulate**
  - Every object has access to its own address through a pointer called **`this`** (a C++ keyword)
  - An object's **`this`** pointer is not part of the object itself
  - The **`this`** pointer is passed (by the compiler) as an implicit argument to each of the object's non-**`static`** member functions
- **Objects use the `this` pointer implicitly or explicitly**
  - Implicitly when accessing members directly
  - Explicitly when using keyword **`this`**
  - Type of the **`this`** pointer depends on the type of the object and whether the executing member function is declared **`const`**



## Outline

fig10\_17.cpp

(1 of 2)

```
1 // Fig. 10.17: fig10_17.cpp
2 // Using the this pointer to refer to object members.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 class Test
8 {
9 public:
10     Test(int = 0); // default constructor
11     void print() const;
12 private:
13     int x;
14 }; // end class Test
15
16 // constructor
17 Test::Test(int value )
18     : x( value ) // initialize x to value
19 {
20     // empty body
21 } // end constructor Test
```



## Outline

fig10\_17.cpp

(2 of 2)

```

22
23 // print x using implicit and explicit this pointers;
24 // the parentheses around *this are required
25 void Test::print() const
26 {
27 // implicitly use the this pointer to access the member x
28 cout << "    " << x;
29
30 // explicitly use the this pointer and
31 // to access the member x
32 cout << "\n this->x = " << this->x;
33
34 // explicitly use the dereferenced this pointer
35 // the dot operator to access the member x
36 cout << "\n(*this).x = " << (*this).x << endl;
37 } // end function print
38
39 int main()
40 {
41     Test testObject(2); // instantiate and initialize testObject
42
43     testObject.print();
44     return 0;
45 } // end main

```

Implicitly using the **this** pointer to access member **x**

Explicitly using the **this** pointer to access member **x**

Using the dereferenced **this** pointer and the dot operator

```

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

```



## Common Programming Error 10.7

---

**Attempting to use the member selection operator (.) with a pointer to an object is a compilation error—the dot member selection operator may be used only with an *lvalue* such as an object's name, a reference to an object or a dereferenced pointer to an object.**



## 10.5 Using the `this` Pointer (Cont.)

- **Cascaded member-function calls**
  - Multiple functions are invoked in the same statement
  - Enabled by member functions returning the dereferenced **`this`** pointer
  - Example
    - `t.setMinute( 30 ).setSecond( 22 );`
      - Calls `t.setMinute( 30 );`
      - Then calls `t.setSecond( 22 );`



## Outline

Time.h

(1 of 2)

```
1 // Fig. 10.18: Time.h
2 // Cascading member function calls.
3
4 // Time class definition.
5 // Member functions defined in Time.cpp.
6 #ifndef TIME_H
7 #define TIME_H
8
9 class Time
10 {
11 public
12     Time(int = 0, int = 0, int = 0); // default constructor
13
14     // set functions (the Time & return types enable cascading)
15     Time &setTime( int, int, int ); // set hour, minute, second
16     Time &setHour( int ); // set hour
17     Time &setMinute( int ); // set minute
18     Time &setSecond( int ); // set second
```

*set* functions return **Time** & to enable cascading



## Outline

Time.h

(2 of 2)

```
19
20 // get functions (normally declared const)
21 int getHour() const; // return hour
22 int getMinute() const; // return minute
23 int getSecond() const; // return second
24
25 // print functions (normally declared const)
26 void printUniversal() const; // print universal time
27 void printStandard() const; // print standard time
28 private
29 int hour; // 0 - 23 (24-hour clock format)
30 int minute; // 0 - 59
31 int second; // 0 - 59
32 }; // end class Time
33
34 #endif
```



## Outline

Time.cpp

(1 of 3)

```
1 // Fig. 10.19: Time.cpp
2 // Member-function definitions for Time class.
3 #include <iostream>
4 using std::cout;
5
6 #include <iomanip>
7 using std::setfill;
8 using std::setw;
9
10 #include "Time.h" // Time class definition
11
12 // constructor function to initialize private data;
13 // calls member function setTime to set variables;
14 // default values are 0 (see class definition)
15 Time::Time( int hr, int min, int sec )
16 {
17     setTime( hr, min, sec );
18 } // end Time constructor
19
20 // set values of hour, minute, and second
21 Time &Time::setTime( int h, int m, int s ) // note Time & return
22 {
23     setHour( h );
24     setMinute( m );
25     setSecond( s );
26     return *this; // enables cascading
27 } // end function setTime
```

Returning dereferenced **this** pointer enables cascading



## Outline

Time.cpp

(2 of 3)

```
28
29 // set hour value
30 Time &Time::setHour( int h )// note Time & return
31 {
32     hour = ( h >= 0 && h < 24 ) ? h : 0; // validate hour
33     return *this; // enables cascading
34 } // end function setHour
35
36 // set minute value
37 Time &Time::setMinute( int m ) // note Time & return
38 {
39     minute = ( m >= 0 && m < 60 ) ? m : 0; // validate minute
40     return *this; // enables cascading
41 } // end function setMinute
42
43 // set second value
44 Time &Time::setSecond( int s ) // note Time & return
45 {
46     second = ( s >= 0 && s < 60 ) ? s : 0; // validate second
47     return *this; // enables cascading
48 } // end function setSecond
49
50 // get hour value
51 int Time::getHour() const
52 {
53     return hour;
54 } // end function getHour
```



## Outline

Time.cpp

(3 of 3)

```
55
56 // get minute value
57 int Time::getMinute() const
58 {
59     return minute;
60 } // end function getMinute
61
62 // get second value
63 int Time::getSecond() const
64 {
65     return second;
66 } // end function getSecond
67
68 // print Time in universal-time format (HH:MM:SS)
69 void Time::printUniversal() const
70 {
71     cout << setfill( '0' ) << setw( 2 ) << hour << ":"
72         << setw( 2 ) << minute << ":" << setw( 2 ) << second;
73 } // end function printUniversal
74
75 // print Time in standard-time format (HH:MM:SS AM or PM)
76 void Time::printStandard() const
77 {
78     cout << ( ( hour == 0 || hour == 12 ) ? 12 : hour % 12 )
79         << " " << setfill( '0' ) << setw( 2 ) << minute
80         << " " << setw( 2 ) << second << ( hour < 12 ? " AM" : " PM" );
81 } // end function printStandard
```



## Outline

fig10\_20.cpp

(1 of 2)

```
1 // Fig. 10.20: fig10_20.cpp
2 // Cascading member function calls with the this pointer.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 #include "Time.h" // Time class definition
8
9 int main()
10 {
11     Time t // create Time object
12
13     // cascaded function calls
14     t.setHour(18).setMinute(30).setSecond(22);
15
16     // output time in universal and standard formats
17     cout << "Universal time: ";
18     t.printUniversal();
19
20     cout << "\nStandard time: ";
21     t.printStandard();
22
23     cout << "\n\nNew standard time: ";
24
25     // cascaded function calls
26     t.setTime(20, 20, 20).printStandard();
27     cout << endl;
28     return 0;
29 } // end main
```

Cascaded function calls using the reference returned by one function call to invoke the next

Note that these calls must appear in the order shown, because `printStandard` does not return a reference to `t`



# Outline

**Universal time: 18:30:22**  
**Standard time: 6:30:22 PM**

**New standard time: 8:20:20 PM**

fig10\_20.cpp

(2 of 2)



# 10.6 Dynamic Memory Management with Operators `new` and `delete`

- **Dynamic memory management**
  - Enables programmers to allocate and deallocate memory for any built-in or user-defined type
  - Performed by operators `new` and `delete`
  - For example, dynamically allocating memory for an array instead of using a fixed-size array



# 10.6 Dynamic Memory Management with Operators `new` and `delete` (Cont.)

- **Operator `new`**

- Allocates (i.e., reserves) storage of the proper size for an object at execution time
- Calls a constructor to initialize the object
- Returns a pointer of the type specified to the right of `new`
- Can be used to dynamically allocate any fundamental type (such as `int` or `double`) or any class type

- **Free store**

- Sometimes called the heap
- Region of memory assigned to each program for storing objects created at execution time



# 10.6 Dynamic Memory Management with Operators new and delete (Cont.)

- **Operator delete**

- Destroys a dynamically allocated object
- Calls the destructor for the object
- Deallocates (i.e., releases) memory from the free store
- The memory can then be reused by the system to allocate other objects



## 10.6 Dynamic Memory Management with Operators `new` and `delete` (Cont.)

- **Initializing an object allocated by `new`**

- **Initializer for a newly created fundamental-type variable**

- **Example**

- `double *ptr = new double( 3.14159 );`

- **Specify a comma-separated list of arguments to the constructor of an object**

- **Example**

- `Time *timePtr = new Time( 12, 45, 0 );`



## Common Programming Error 10.8

---

**Not releasing dynamically allocated memory when it is no longer needed can cause the system to run out of memory prematurely. This is sometimes called a “**memory leak.**”**



## 10.6 Dynamic Memory Management with Operators `new` and `delete` (Cont.)

- **`new` operator can be used to allocate arrays dynamically**
  - Dynamically allocate a 10-element integer array:  

```
int *gradesArray = new int[ 10 ];
```
  - Size of a dynamically allocated array
    - Specified using any integral expression that can be evaluated at execution time



## 10.6 Dynamic Memory Management with Operators new and delete (Cont.)

- **Delete a dynamically allocated array:**

```
delete [] gradesArray;
```

- This deallocates the array to which **gradesArray** points
- If the pointer points to an array of objects
  - First calls the destructor for every object in the array
  - Then deallocates the memory
- If the statement did not include the square brackets (**[ ]**) and **gradesArray** pointed to an array of objects
  - Only the first object in the array would have a destructor call



## Common Programming Error 10.9

---

Using **delete** instead of **delete []** for arrays of objects can lead to runtime logic errors. To ensure that every object in the array receives a destructor call, always delete memory allocated as an array with operator **delete []**. Similarly, always delete memory allocated as an individual element with operator **delete**.



# 10.7 static Class Members

- **static data member**
  - Only one copy of a variable shared by all objects of a class
    - “Class-wide” information
    - A property of the class shared by all instances, not a property of a specific object of the class
  - Declaration begins with keyword **static**



# 10.7 static Class Members (Cont.)

- **static data member (Cont.)**

- **Example**

- **Video game with Martians and other space creatures**
  - **Each Martian needs to know the martianCount**
  - **martianCount should be static class-wide data**
  - **Every Martian can access martianCount as if it were a data member of that Martian**
  - **Only one copy of martianCount exists**
- **May seem like global variables but have class scope**
- **Can be declared public, private or protected**



# 10.7 static Class Members (Cont.)

- **static data member (Cont.)**

- **Fundamental-type static data members**
  - **Initialized by default to 0**
  - **If you want a different initial value, a static data member can be initialized once (and only once)**
- **A const static data member of int or enum type**
  - **Can be initialized in its declaration in the class definition**
- **All other static data members**
  - **Must be defined at file scope (i.e., outside the body of the class definition)**
  - **Can be initialized only in those definitions**
- **static data members of class types (i.e., static member objects) that have default constructors**
  - **Need not be initialized because their default constructors will be called**



# 10.7 static Class Members (Cont.)

- **static data member (Cont.)**

- **Exists even when no objects of the class exist**
  - **To access a public static class member when no objects of the class exist**
    - **Prefix the class name and the binary scope resolution operator (::) to the name of the data member**
    - **Example**
      - **Martian::martianCount**
- **Also accessible through any object of that class**
  - **Use the object's name, the dot operator and the name of the member**
    - **Example**
      - **myMartian.martianCount**



## 10.7 `static` Class Members (Cont.)

- **`static` member function**
  - Is a service of the *class*, not of a specific object of the class
- **`static` applied to an item at file scope**
  - That item becomes known only in that file
  - The **`static`** members of the class need to be available from any client code that accesses the file
    - So we cannot declare them **`static`** in the `.cpp` file—we declare them **`static`** only in the `.h` file.



## Performance Tip 10.3

---

Use **static** data members to save storage when a single copy of the data for all objects of a class will suffice.



## Software Engineering Observation 10.10

---

**A class's `static` data members and `static` member functions exist and can be used even if no objects of that class have been instantiated.**



# Common Programming Error 10.10

---

**It is a compilation error to include keyword `static` in the definition of a `static` data members at file scope.**



## Outline

fig10\_21.cpp

(1 of 1)

```
1 // Fig. 10.21: Employee.h
2 // Employee class definition.
3 #ifndef EMPLOYEE_H
4 #define EMPLOYEE_H
5
6 class Employee
7 {
8 public
9     Employee(const char * const, const char * const ); // constructor
10    ~Employee(); // destructor
11    const char *getFirstName() const; // return first name
12    const char *getLastName() const; // return last name
13
14    // static member function
15    static int getCount(); // return number of objects instantiated
16 private
17    char *firstName;
18    char *lastName;
19
20    // static data
21    static int count; // number of objects instantiated
22 }; // end class Employee
23
24 #endif
```

Function prototype for **static** member function

**static** data member keeps track of number of **Employee** objects that currently exist



## Outline

Employee.cpp

(1 of 3)

```
1 // Fig. 10.22: Employee.cpp
2 // Member-function definitions for class Employee.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 #include <cstring> // strlen and strcpy prototypes
8 using std::strlen;
9 using std::strcpy;
10
11 #include "Employee.h" // Employee class definition
12
13 // define and initialize static data member at file scope
14 in Employee::count = 0;
15
16 // define static member function
17 // Employee objects instantiated
18 in Employee::getCount()
19 {
20     return count;
21 } // end static function getCount
```

**static** data member is defined and initialized at file scope in the **.cpp** file

**static** member function can access only **static** data, because the function might be called when no objects exist



## Outline

```

22
23 // constructor dynamically allocates space for first and last name and
24 // uses strcpy to copy first and last names into the object
25 Employee::Employee( const char * const first, const char * const last )
26 {
27     firstName = new char[ strlen( first ) + 1 ];
28     strcpy( firstName, first );
29
30     lastName = new char[ strlen( last ) + 1 ];
31     strcpy( lastName, last );
32
33     count++; // increment static count of employees
34
35     cout << "Employee constructor for " << firstName
36         << << lastName << " called." << endl;
37 } // end Employee constructor
38
39 // destructor deallocates dynamically allocated memory
40 Employee::~Employee()
41 {
42     cout << "~Employee() called for " << firstName
43         << << lastName << endl;
44
45     delete [] firstName; // release memory
46     delete [] lastName; // release memory
47
48     count--; // decrement static count of employees
49 } // end ~Employee destructor

```

Employee.cpp

Dynamically allocating **char** arrays

Non-**static** member function (i.e., constructor) can modify the class's **static** data members

Deallocating memory reserved for arrays



## Outline

Employee.cpp

(3 of 3)

```
50
51 // return first name of employee
52 const char*Employee::getFirstName() const
53 {
54     // const before return type prevents client from modifying
55     // private data; client should copy returned string before
56     // destructor deletes storage to prevent undefined pointer
57     return firstName;
58 } // end function getFirstName
59
60 // return last name of employee
61 const char*Employee::getLastName() const
62 {
63     // const before return type prevents client from modifying
64     // private data; client should copy returned string before
65     // destructor deletes storage to prevent undefined pointer
66     return lastName;
67 } // end function getLastName
```



## Outline

fig10\_23.cpp

(1 of 2)

```

1 // Fig. 10.23: fig10_23.cpp
2 // Driver to test class Employee.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 #include "Employee.h" // Employee class definition
8
9 int main()
10 {
11 // use class name and binary scope resolution operator to
12 // access static number function getCount
13 cout << "Number of employees before instantiation of any objects is "
14     << Employee::getCount() << endl; // use class name
15
16 // use new to dynamically create two new Employees
17 // operator new also calls the object's constructor
18 Employee *e1Ptr = new Employee( "Susan", "Baker" );
19 Employee *e2Ptr = new Employee( "Robert", "Jones" );
20
21 // call getCount on first Employee object
22 cout << "Number of employees after objects are inst
23     << e1Ptr->getCount();
24
25 cout << "\n\nEmployee 1: "
26     << e1Ptr->getFirstName() << " " << e1Ptr
27     << "\nEmployee 2: "
28     << e2Ptr->getFirstName() << " " << e2Ptr->getLastName() << "\n\n";

```

Calling **static** member function using class name and binary scope resolution operator

Dynamically creating **Employees** with **new**

Calling a **static** member function through a pointer to an object of the class



## Outline

```

29
30 delete e1Ptr; // deallocate memory
31 e1Ptr = 0; // disconnect pointer from free-store space
32 delete e2Ptr; // deallocate memory
33 e2Ptr = 0; // disconnect pointer from free-store space
34
35 // no objects exist, so call static method
36 // using the class name and the binary scope resolution operator
37 cout << "Number of employees after objects are deleted is "
38     << Employee::getCount() << endl;
39 return 0;
40 } // end main

```

Releasing memory to which a pointer points

Disconnecting a pointer from any space in memory

fig10\_23.cpp

(2 of 2)

```

Number of employees before instantiation of any objects is 0
Employee constructor for Susan Baker called.
Employee constructor for Robert Jones called.
Number of employees after objects are instantiated is 2

Employee 1: Susan Baker
Employee 2: Robert Jones

~Employee() called for Susan Baker
~Employee() called for Robert Jones
Number of employees after objects are deleted is 0

```



## 10.7 static Class Members (Cont.)

- **Declare a member function static**
  - If it does not access **non-static** data members or **non-static** member functions of the class
  - A **static** member function does not have a **this** pointer
  - **static** data members and **static** member functions exist independently of any objects of a class
  - When a **static** member function is called, there might not be any objects of its class in memory



# Software Engineering Observation 10.11

---

**Some organizations specify in their software engineering standards that all calls to `static` member functions be made using the class name and not an object handle.**



# Common Programming Error 10.11

---

Using the **this** pointer in a **static** member function is a compilation error.



## Common Programming Error 10.12

---

Declaring a **static** member function **const** is a compilation error. The **const** qualifier indicates that a function cannot modify the contents of the object in which it operates, but **static** member functions exist and operate independently of any objects of the class.



## Error-Prevention Tip 10.2

---

**After deleting dynamically allocated memory, set the pointer that referred to that memory to  $\mathbf{0}$ . This disconnects the pointer from the previously allocated space on the free store. This space in memory could still contain information, despite having been deleted. By setting the pointer to  $\mathbf{0}$ , the program loses any access to that free-store space, which, in fact, could have already been reallocated for a different purpose. If you didn't set the pointer to  $\mathbf{0}$ , your code could inadvertently access this new information, causing extremely subtle, nonrepeatable logic errors.**

---



# 10.8 Data Abstraction and Information Hiding

- **Information Hiding**

- A class normally hides implementation details from clients

- **Data abstraction**

- Client cares about *what* functionality a class offers, not about *how* that functionality is implemented
  - For example, a client of a stack class need not be concerned with the stack's implementation (e.g., a linked list)
- Programmers should not write code that depends on a class's implementation details



# 10.8 Data Abstraction and Information Hiding (Cont.)

- **Importance of data**

- **Elevated in C++ and object-oriented community**
  - **Primary activities of object-oriented programming in C++**
    - **Creation of types (i.e., classes)**
    - **Expression of the interactions among objects of those types**
- **Abstract data types (ADTs)**
  - **Improve the program development process**



# 10.8 Data Abstraction and Information Hiding (Cont.)

- **Abstract data types (ADTs)**
  - **Essentially ways of representing real-world notions to some satisfactory level of precision within a computer system**
  - **Types like `int`, `double`, `char` and others are all ADTs**
    - **e.g., `int` is an abstract representation of an integer**
  - **Capture two notions:**
    - **Data representation**
    - **Operations that can be performed on the data**
  - **C++ classes implement ADTs and their services**



## 10.8.1 Example: Array Abstract Data Type

- **Many array operations not built into C++**
  - e.g., subscript range checking
- **Programmers can develop an array ADT as a class that is preferable to “raw” arrays**
  - Can provide many helpful new capabilities
- **C++ Standard Library class template `vector`**



## Software Engineering Observation 10.12

---

**The programmer is able to create new types through the class mechanism. These new types can be designed to be used as conveniently as the built-in types. Thus, C++ is an extensible language. Although the language is easy to extend with these new types, the base language itself cannot be changed.**



## 10.8.2 Example: String Abstract Data Type

- **No string data type among C++'s built-in data types**
  - **C++ is an intentionally sparse language**
    - **Provides programmers with only the raw capabilities needed to build a broad range of systems**
    - **Designed to minimize performance burdens**
    - **Designed to include mechanisms for creating and implementing string abstract data types through classes**
  - **C++ Standard Library class `string`**



## 10.8.3 Example: Queue Abstract Data Type

- **Queue ADT**
  - **Items returned in first-in, first-out (FIFO) order**
    - **First item inserted in the queue is the first item removed from the queue**
  - **Hides an internal data representation that somehow keeps track of the items currently waiting in line**
  - **Good example of an abstract data type**
    - **Clients invoke *enqueue* operation to put things in the queue one at a time**
    - **Clients invoke *dequeue* operation to get those things back one at a time on demand**
  - **C++ Standard Library `queue` class**



# 10.9 Container Classes and Iterators

- **Container classes (also called collection classes)**
  - **Classes designed to hold collections of objects**
  - **Commonly provide services such as insertion, deletion, searching, sorting, and testing an item to determine whether it is a member of the collection**
  - **Examples**
    - **Arrays**
    - **Stacks**
    - **Queues**
    - **Trees**
    - **Linked lists**



# 10.9 Container Classes and Iterators (Cont.)

- **Iterator objects—or more simply iterators**
  - Commonly associated with container classes
  - An object that “walks through” a collection, returning the next item (or performing some action on the next item)
  - A container class can have several iterators operating on it at once
  - Each iterator maintains its own “position” information



## 10.10 Proxy Classes

- **Header files contain some portion of a class's implementation and hints about others**
  - For example, a class's **private** members are listed in the class definition in a header file
  - Potentially exposes proprietary information to clients of the class



## 10.10 Proxy Classes (Cont.)

- **Proxy class**
  - **Hides even the `private` data of a class from clients**
  - **Knows only the `public` interface of your class**
  - **Enables the clients to use your class's services without giving the client access to your class's implementation details**



## Outline

## Implementation.h

(1 of 1)

```
1 // Fig. 10.24: Implementation.h
2 // Header file for class Implementation
3
4 class Implementation
5 {
6 public
7     // constructor
8     Implementation(int v )
9         : value(v) // initialize value with v
10    {
11        // empty body
12    } // end constructor Implementation
13
14    // set value to v
15    void setValue( int v )
16    {
17        value = v; // should validate v
18    } // end function setValue
19
20    // return value
21    int getValue() const
22    {
23        return value;
24    } // end function getValue
25 private:
26    int value; // data that we would like to hide from the client
27 }; // end class Implementation
```

Class definition for the class that contains the proprietary implementation we would like to hide

The data we would like to hide from the client



## Outline

## Interface.h

(1 of 1)

```
1 // Fig. 10.25: Interface.h
2 // Header file for class Interface
3 // Client sees this source code, but the source code does not reveal
4 // the data layout of class Implementation.
5
6 class Implementation; // forward class declaration required by line 17
7
8 class Interface
9 {
10 public
11     Interface(t); // constructor
12     void setValue(int); // same public interface as
13     int getValue() const; // class Implementation has
14     ~Interface(); // destructor
15 private
16     // requires previous forward declaration (line 6)
17     Implementation *ptr;
18 }; // end class Interface
```

Declares **Implementation** as a data type without including the class's complete header file

**public** interface between client and hidden class

Using a pointer allows us to hide implementation details of class **Implementation**



## Outline

Interface.cpp

(1 of 1)

```

1 // Fig. 10.26: Interface.cpp
2 // Implementation of class Interface--client receives this file only
3 // as precompiled object code, keeping the implementation hidden.
4 #include "Interface.h" Interface class definition
5 #include "Implementation.h" // Implementation class definition
6
7 // constructor
8 Interface::Interface( int v )
9     : ptr( new Implementation( v ) ) //
10 {
11     // empty body
12 } // end Interface constructor
13
14 // call Implementation's setValue function
15 void Interface::setValue( int v )
16 {
17     ptr->setValue( v );
18 } // end function setValue
19
20 // call Implementation's getValue function
21 int Interface::getValue() const
22 {
23     return ptr->getValue();
24 } // end function getValue
25
26 // destructor
27 Interface::~Interface()
28 {
29     delete ptr;
30 } // end ~Interface destructor

```

Only location where **Implementation.h**  
is included with **#include**

Setting the value of the hidden data via a pointer

Getting the value of the hidden data via a pointer



## Outline

fig10\_27.cpp

(1 of 1)

```
1 // Fig. 10.27: fig10_27.cpp
2 // Hiding a class's private data with a proxy class.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 #include "Interface" // Interface class definition
8
9 int main()
10 {
11     Interface i(5); // create Interface
12
13     cout << "Interface contains: " << i.getValue()
14         << "before setValue" << endl;
15
16     i.setValue(10);
17
18     cout << "Interface contains: " << i.getValue()
19         << "after setValue" << endl;
20     return 0;
21 } // end main
```

Only the header file for **Interface** is included in the client code—no mention of the existence of a separate class called **Implementation**

```
Interface contains: 5 before setValue
Interface contains: 10 after setValue
```



# Software Engineering Observation 10.13

---

**A proxy class insulates client code from implementation changes.**

