### Capitolo 7: Classes Part II

### **Outline**

7.1	Introduction
7.2	const (Constant) Objects and const Member Functions
7.3	Composition: Objects as Members of Classes
7.4	friend Functions and friend Classes
7.5	Using the this Pointer
7.6	<b>Dynamic Memory Allocation with Operators new and delete</b>
7.7	static Class Members
7.8	Data Abstraction and Information Hiding
7.8.1	Example: Array Abstract Data Type
7.8.2	Example: String Abstract Data Type
7.8.3	Example: Queue Abstract Data Type
7.9	Container Classes and Iterators
7.10	Proxy Classes



### 7.1 Introduction

- Chapters 6 through 8 discuss object-based programming (OBP)
- Chapters 9 and 10 discuss inheritance and polymorphism

## 7.2 const (Constant) Objects and const Member Functions

- Principle of least privilege
  - Only give objects permissions they need, no more
- Keyword const
  - Specify that an object is not modifiable
  - Any attempt to modify the object is a syntax error
  - Example

```
const Time noon( 12, 0, 0 );
```

• Declares a const object noon of class **Time** and initializes it to 12



## 7.2 const (Constant) Objects and const Member Functions

- const objects require const functions
  - Member functions declared **const** cannot modify their object
  - const must be specified in function prototype and definition
  - Prototype:

```
ReturnType FunctionName(param1,param2...) const;
```

– Definition:

```
ReturnType FunctionName(param1,param2...) const { ...}
```

– Example:

```
int A::getValue() const { return
  privateDataMember };
```

- Returns the value of a data member but doesn't modify anything so is declared **const**
- Constructors / Destructors cannot be const
  - They need to initialize variables, therefore modifying them



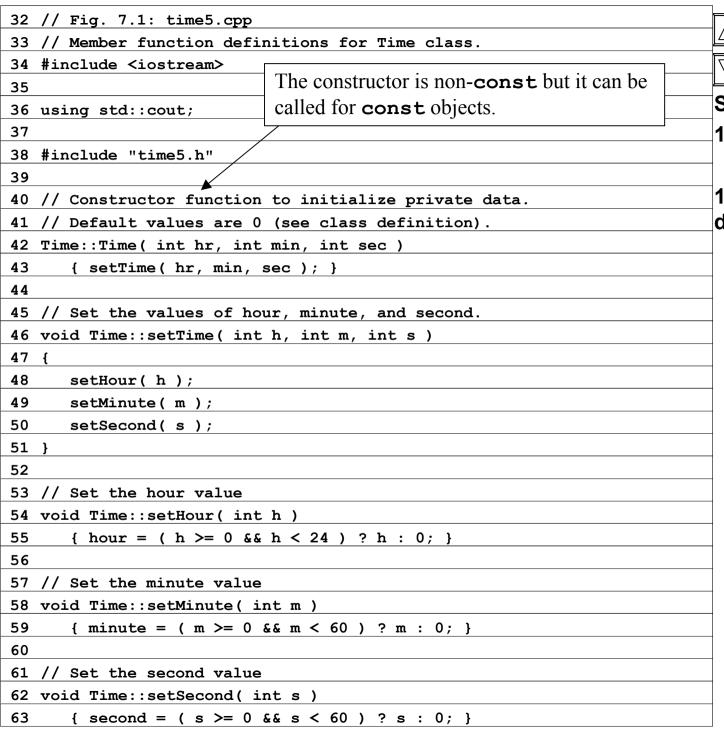
```
1 // Fig. 7.1: time5.h
2 // Declaration of the class Time.
3 // Member functions defined in time5.cpp
  #ifndef TIME5 H
5 #define TIME5 H
6
7 class Time {
8 public:
      Time( int = 0, int = 0, int = 0 ); // default constructor
9
10
11
      // set functions
                                      // set time
12
      void setTime( int, int, int );
13
      void setHour( int );
                               // set hour
                               // set minute
14
      void setMinute( int );
                               // set_second
      void setSecond( int );
15
16
      // get functions (normally declared const)
17
18
      int getHour() const;
                               // return hour
19
      int getMinute() const;
                               // return minute
                                                          const
20
      int getSecond() const;
                               // return second
                                                          functions
21
      // print functions (normally declared const)
22
23
      void printMilitary() const; // print military time
24
      void printStandard();
                                   // print standard time
                                                          non-const
25 private:
                                                          functions
26
      int hour;
                             // 0 - 23
                             // 0 - 59
27
      int minute;
                             // 0 - 59
28
      int second;
29 };
30
31 #endif
```



#### 1. Class definition

### 1.1 Function prototypes

#### 1.2 Member variables



<u>Outline</u>

Source Code

1. Load Header

1.1 Function definitions

88 }

```
89 // Fig. 7.1: fig07 01.cpp
                                                                         Outline
90 // Attempting to access a const object with
91 // non-const member functions.
92 #include "time5.h"
                                                                  1 Initialize variables
93
94 int main()
95 {
     Time wakeUp(6, 45, 0);  // non-constant object
96
                                                                  2. Attempt to use non-
     const Time noon( 12, 0, 0 ); // constant object
97
                                                                  const functions with
98
                                                                  const objects
99
                          // MEMBER FUNCTION OBJECT
     wakeUp.setHour( 18 ); // non-const
100
                                             non-const
101
102
     noon.setHour( 12 ); // non-const
                                             const
103
     104
                                             non-const
105
     noon.getMinute();  // const
106
                                             const
     noon.printMilitary(); // const
107
                                             const
     noon.printStandard(); // non-const
108
                                             const
109 return 0;
                                                           Compiler errors generated.
110}
```

```
Compiling...

Fig07_01.cpp
d:fig07_01.cpp(14) : error C2662: 'setHour' : cannot convert 'this'
pointer from 'const class Time' to 'class Time &'
Conversion loses qualifiers
d:\fig07_01.cpp(20) : error C2662: 'printStandard' : cannot convert
'this' pointer from 'const class Time' to 'class Time &'
Conversion loses qualifiers
Time5.cpp
Error executing cl.exe.

test.exe - 2 error(s), 0 warning(s)
```

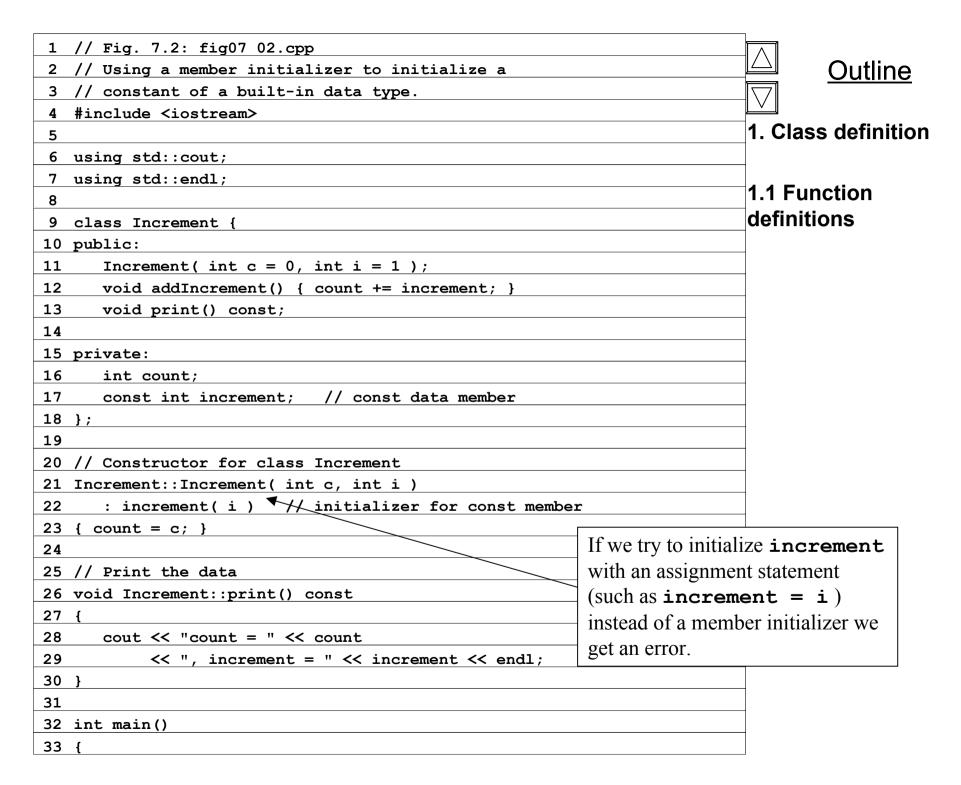
#### **Program Output**

## 7.2 const (Constant) Objects and const Member Functions

- Member initializer syntax
  - Data member increment in class Increment
  - constructor for Increment is modified as follows:

```
Increment::Increment( int c, int i )
    : increment( i )
    { count = c; }
```

- -: increment(i) initializes increment to i
- All data members can be initialized using member initializer syntax
- consts and references must be initialized using member initializer syntax
- Multiple member initializers
  - Use comma-separated list after the colon



```
34
      Increment value( 10, 5 );
35
36
      cout << "Before incrementing: ";</pre>
37
      value.print();
38
39
      for (int j = 0; j < 3; j++) {
40
         value.addIncrement();
41
         cout << "After increment " << j + 1 << ": ";</pre>
         value.print();
42
43
      }
44
45
      return 0;
46 }
```

```
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
```

### **Outline**

- 1.2 Initialize variables
- 2. Function calls
- 3. Output results

# 7.3 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 their enclosing class objects (host objects)

```
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
      void print() const; // print date in month/day/year format
10
      ~Date(); // provided to confirm destruction order
11
12 private:
      int month; // 1-12
13
      int day; // 1-31 based on month
14
      int year; // any year
15
16
17
      // utility function to test proper day for month and year
18
      int checkDay( int );
19 };
20
21 #endif
```

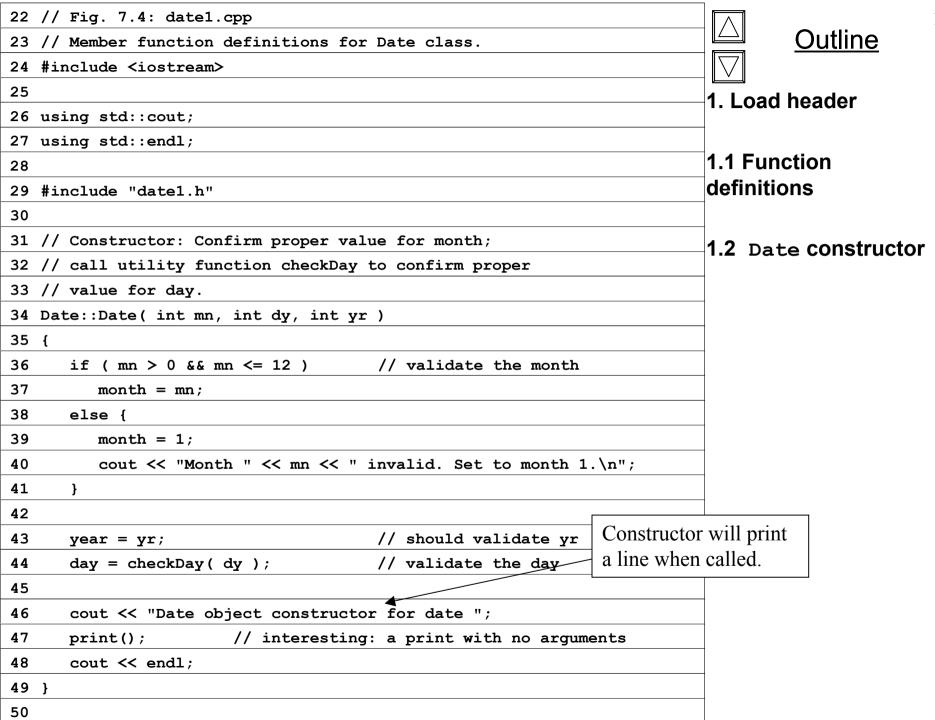
<u>Outline</u>

<u>----</u>

1. Class definition

1.1 Member functions

1.2 Member variables



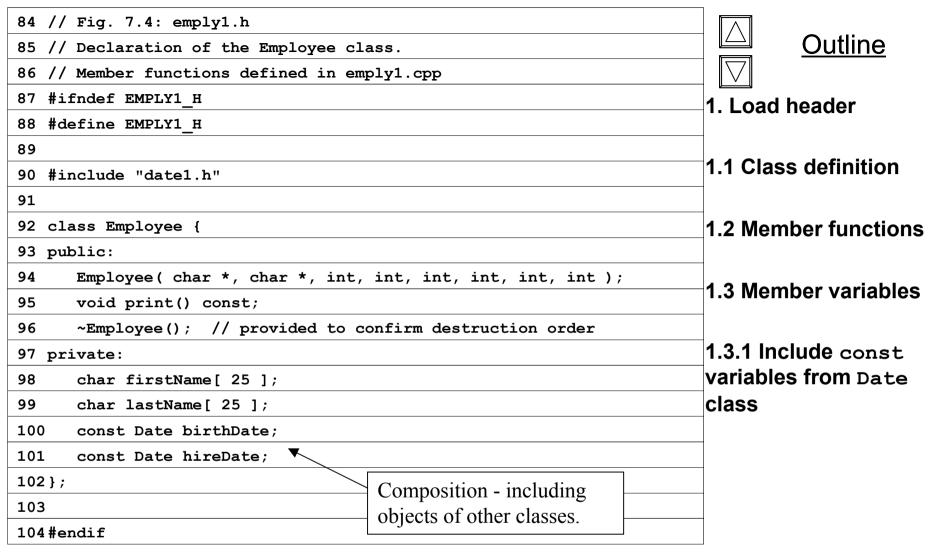
```
51 // Print Date object in form month/day/year
52 void Date::print() const
      { cout << month << '/' << day << '/' << year; }
53
54
                                              Destructor will print
55 // Destructor: provided to confirm destru
                                              a line when called.
56 Date::~Date()
57 {
      cout << "Date object destructor for date ";</pre>
58
59
      print();
      cout << endl;</pre>
60
61 }
62
63 // Utility function to confirm proper day value
64 // based on month and year.
65 // Is the year 2000 a leap year?
66 int Date::checkDay( int testDay )
67 {
      static const int daysPerMonth[ 13 ] =
68
69
         {0, 31, 28, 31, 30, 31, 30, 31, 30, 31, 30, 31};
70
      if ( testDay > 0 && testDay <= daysPerMonth[ month ] )</pre>
71
72
         return testDay;
73
                             // February: Check for leap year
      if ( month == 2 &&
74
75
           testDay == 29 &&
76
           ( year % 400 == 0 ||
77
            ( year % 4 == 0 && year % 100 != 0 ) ))
78
         return testDay;
79
      cout << "Day " << testDay << " invalid. Set to day 1.\n";</pre>
80
81
      return 1; // leave object in consistent state if bad value
82
83 }
```

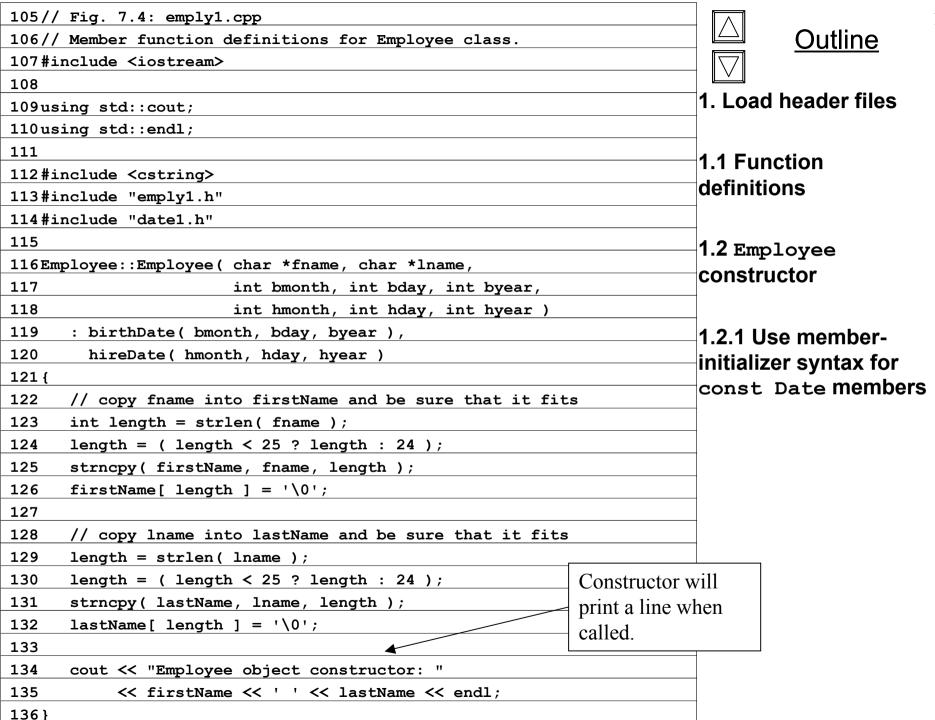
### **Outline**

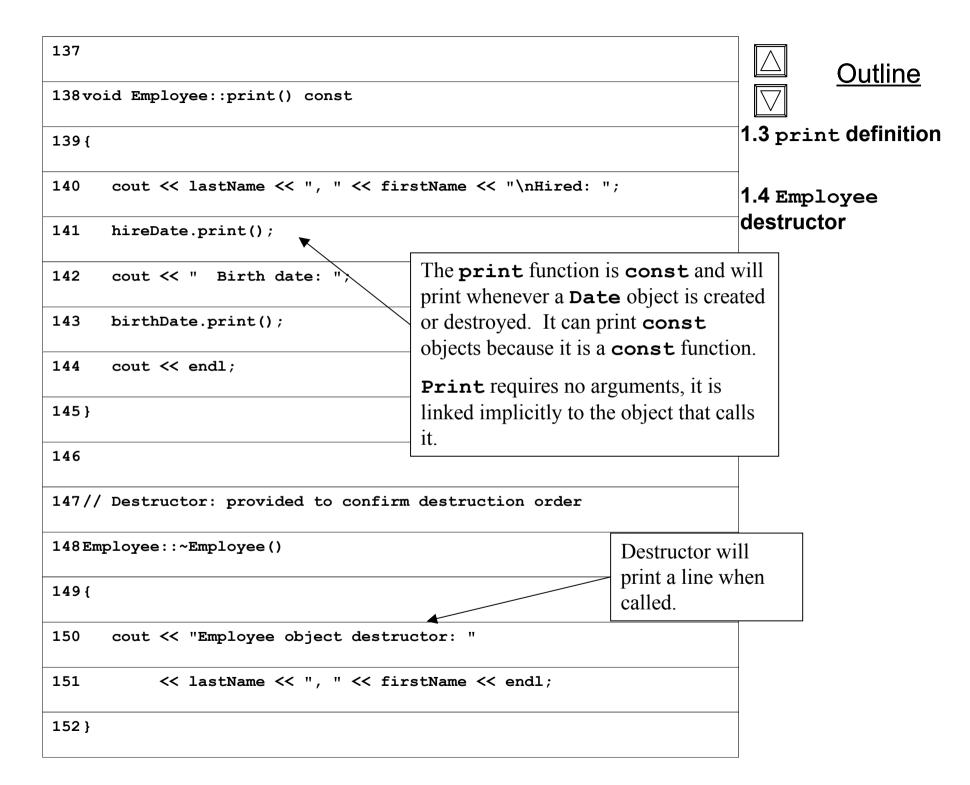
1.3 print function

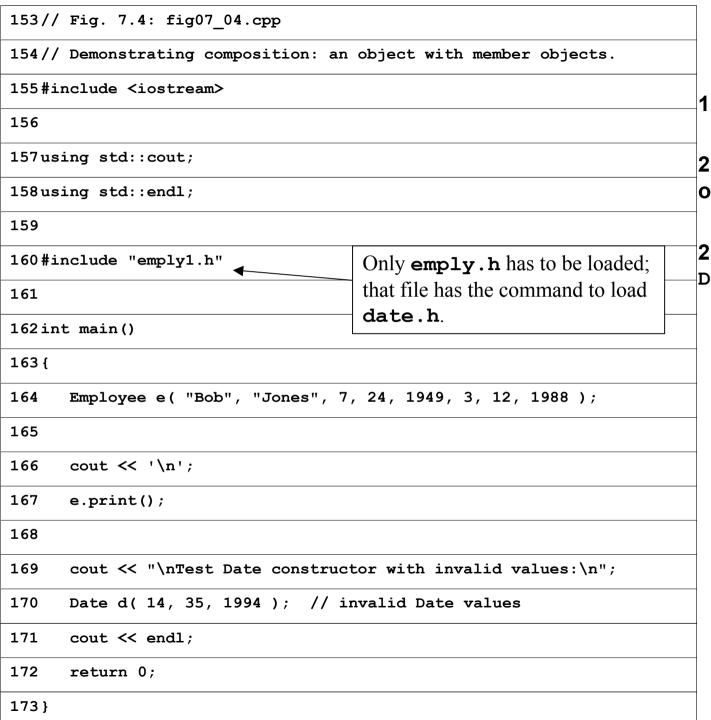
1.4 Date destructor

1.5 checkDay function











### **Outline**

- 1. Load header files
- 2. Create Employee object
- 2.1 Attempt invalid Date setting

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



**Outline** 

**Program Output** 

Notice how inner objects are created first and destroyed last.

### 7.4 friend Functions and friend Classes

- friend function and friend classes
  - Can access private and protected members of another class
  - **friend** functions are not member functions of class
    - Defined outside of class scope
- Properties of friendship
  - Friendship is granted, not taken
  - Not symmetric (if B a friend of A, A not necessarily a friend of B)
  - Not transitive (if A a friend of B, B a friend of C, A not necessarily a friend of C)

### 7.4 friend Functions and friend Classes

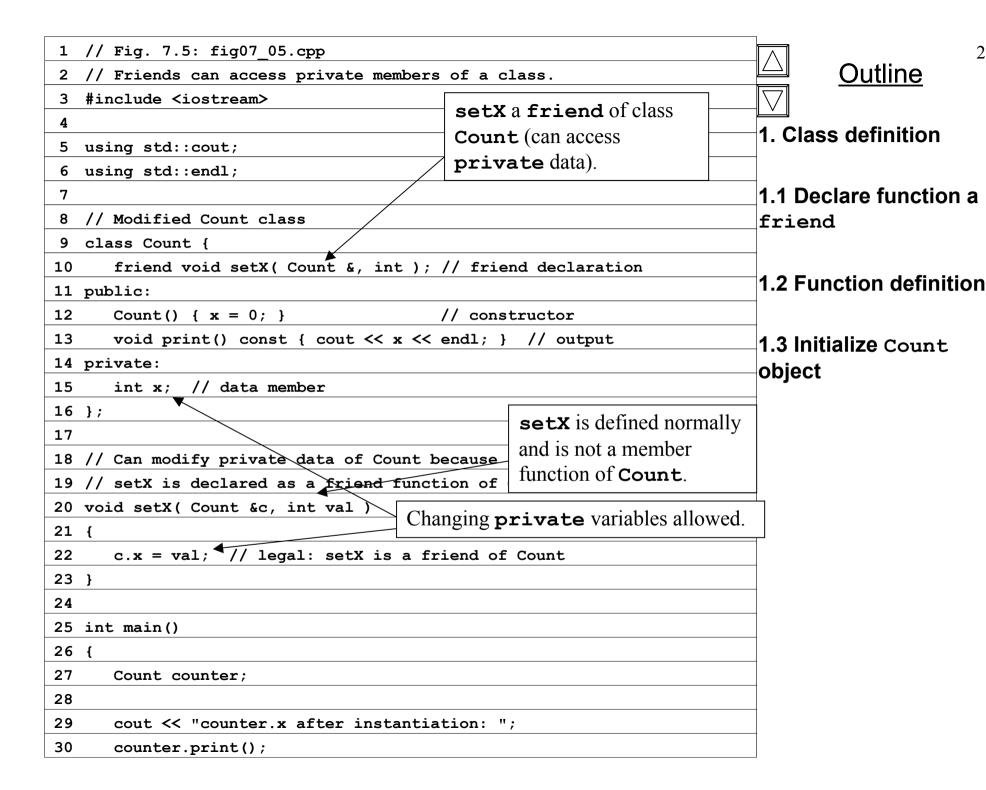
- **friend** declarations
  - To declare a friend function
    - Type **friend** before the function prototype in the class that is giving friendship

```
friend int myFunction( int x );
```

should appear in the class giving friendship

- To declare a **friend** class
- Type friend class Classname in the class that is giving friendship
- if ClassOne is granting friendship to ClassTwo,
   friend class ClassTwo;
- should appear in ClassOne's definition





```
31 cout << "counter.x after call to setX friend function: ";

32 setX( counter, 8 ); // set x with a friend

33 counter.print();

34 return 0;

35 }

36 Dutline

20 Outline

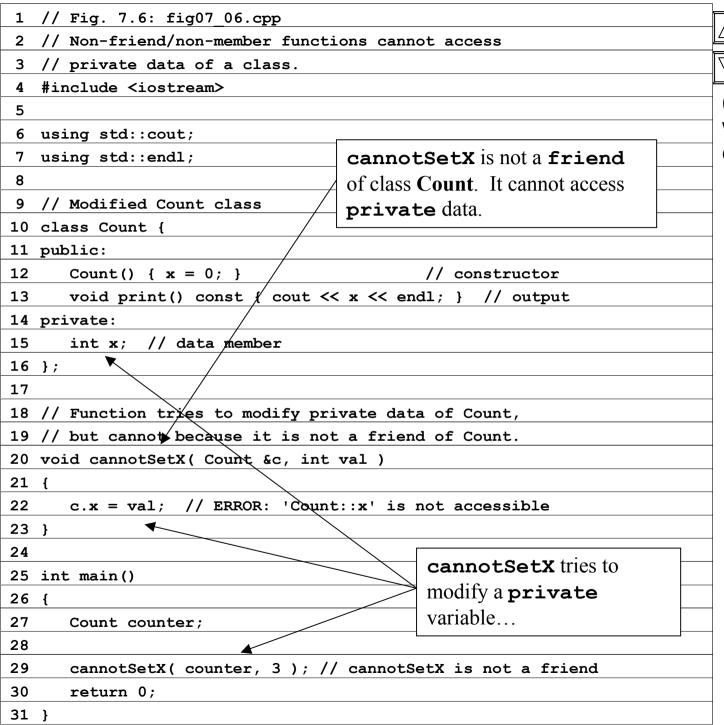
37 Outline

38 A Print results
```

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

Program Output

private data was changed.
```



### <u>Outline</u>

(Previous program without friend declared)

Expected compiler error - cannot

access private data

**Outline** 

**Program Output** 

© 2000 Prentice Hall, Inc. All rights reserved.

### 7.5 Using the this Pointer

### • this pointer

- Allows objects to access their own address
- Not part of the object itself
- Implicit first argument on non-static member function call to the object
- Implicitly reference member data and functions
- The type of the **this** pointer depends upon the type of the object and whether the member function using **this** is **const**
- In a non-const member function of Employee, this has type
   Employee \* const
  - Constant pointer to an **Employee** object
- In a const member function of Employee, this has type
   const Employee \* const
  - Constant pointer to a constant **Employee** object



### 7.5 Using the this Pointer

- Examples using this
  - For a member function print data member x, either

```
or (*this).x
```

- Cascaded member function calls
  - Function returns a reference pointer to the same object { return \*this; }
  - Other functions can operate on that pointer
  - Functions that do not return references must be called last

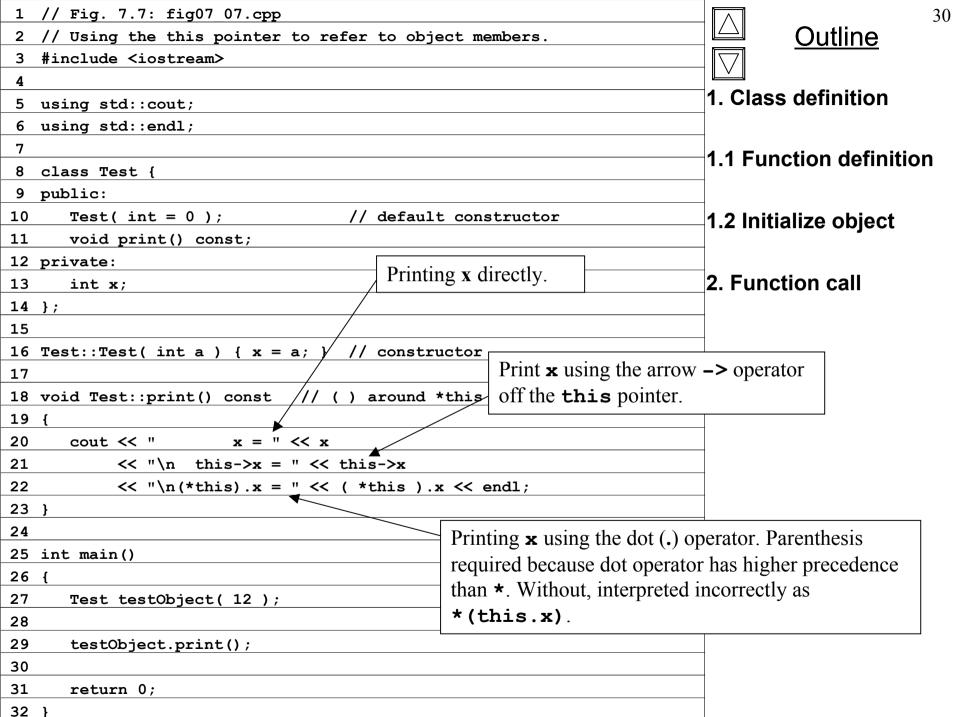
### 7.5 Using the this Pointer

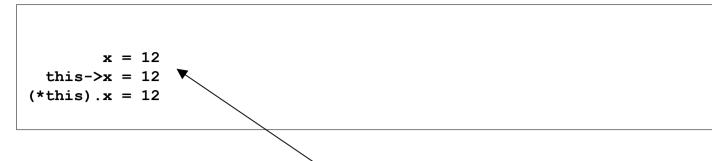
- Example of cascaded member function calls
  - Member functions setHour, setMinute, and setSecond
     all return \*this (reference to an object)
  - For object t, consider
    t.setHour(1).setMinute(2).setSecond(3);
  - Executes t.setHour(1), returns \*this (reference to object) and the expression becomes

```
t.setMinute(2).setSecond(3);
```

- Executes t.setMinute(2), returns reference and becomes t.setSecond(3);
- Executes t.setSecond(3), returns reference and becomest;
- Has no effect







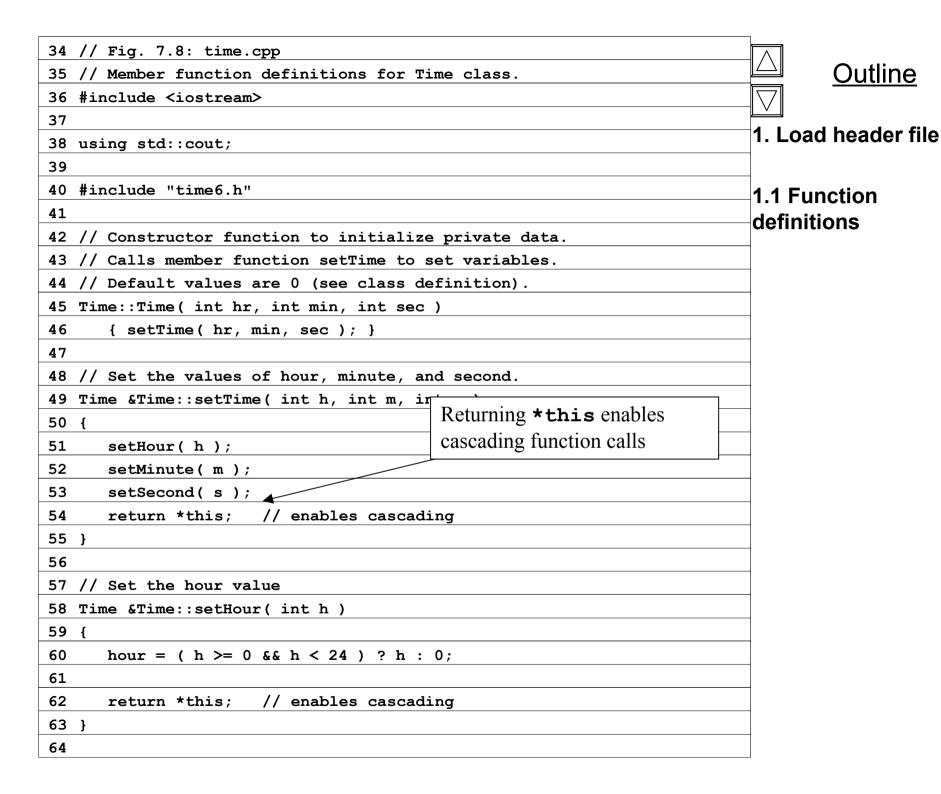
Outline

Program Output

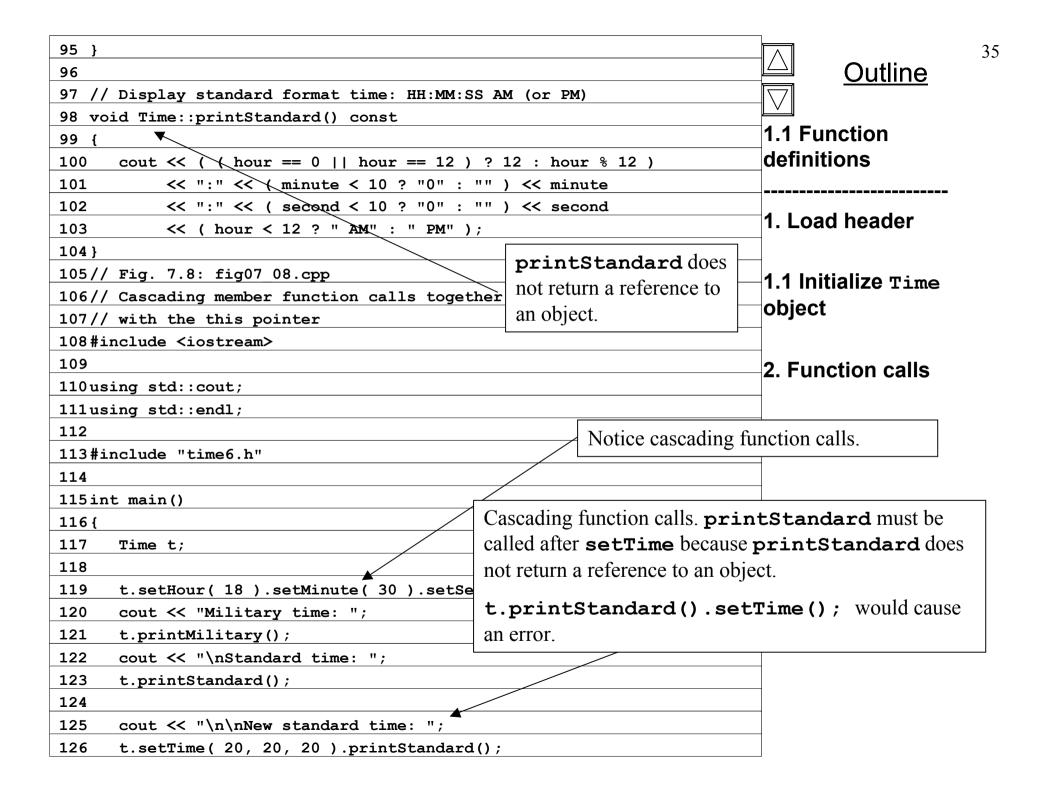
All three methods have the same result.

1 // Fig. 7.8: time6.h	\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
2 // Cascading member function calls.	<u> Outline</u>
3	
4 // Declaration of class Time.	 1. Class definition
- , ,	i. Ciass delillidi
6 #ifndef TIME6 H	
7 #define TIME6 H	
8	
9 class Time {	
10 public:	
<pre>11  Time( int = 0, int = 0, int = 0 ); // default constructor</pre>	
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 min Notice the Time & - function	
returns a reference to a Wime	
10	
19 // get functions (normally declared object. Specify object in	
20 int getHour() const; // return function definition.	
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	

**Outline** 



```
65 // Set the minute value
                                                                                  Outline
66 Time &Time::setMinute( int m )
67 {
      minute = ( m \ge 0 \&\& m < 60 ) ? m : 0;
68
                                                                          1.1 Function
69
                                                                          definitions
70
                     // enables cascading
      return *this;
71 }
72
73 // Set the second value
                                                           Returning *this enables
74 Time &Time::setSecond(int s)
                                                           cascading function calls
75 {
76
      second = (s \ge 0 \&\& s < 60) ? s : 0;
77
78
      return *this; // enables cascading
79 }
80
81 // Get the hour value
82 int Time::getHour() const { return hour; }
83
84 // Get the minute value
85 int Time::getMinute() const { return minute; }
86
87 // Get the second value
88 int Time::getSecond() const { return second; }
89
90 // Display military format time: HH:MM
91 void Time::printMilitary() const
92 {
      cout << ( hour < 10 ? "0" : "" ) << hour << ":"
93
           << ( minute < 10 ? "0" : "" ) << minute;
94
```



127	cout << endl;
128	
129	return 0;
130}	

Outline

7

Military time: 18:30

Standard time: 6:30:22 PM

New standard time: 8:20:20 PM

**Program Output** 

# 7.6 Dynamic Memory Allocation with Operators new and delete

- new and delete
  - Used for dynamic memory allocation
    - Superior to C's malloc and free
  - new
    - Creates an object of the proper size, calls its constructor and returns a pointer of the correct type
  - delete
    - Destroys object and frees space
  - Examples of new

```
TypeName *typeNamePtr;
```

• Creates pointer to a **TypeName** object

```
typeNamePtr = new TypeName;
```

• new creates TypeName object, returns pointer (which typeNamePtr is set equal to)



# 7.6 Dynamic Memory Allocation with Operators new and delete

Examples of delete

```
delete typeNamePtr;
```

- Calls destructor for **TypeName** object and frees memory **Delete** [] **arrayPtr**;
- Used to dynamically delete an array
- Initializing objects

```
double *thingPtr = new double( 3.14159 );
```

Initializes object of type double to 3.14159

```
int *arrayPtr = new int[ 10 ];
```

- Creates a ten element **int** array and assigns it to **arrayPtr** 

### 7.7 static Class Members

#### • static class members

- Shared by all objects of a class
  - Normally, each object gets its own copy of each variable
- Efficient when a 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 at file scope
- Exist even if no instances (objects) of the class exist
- Both variables and functions can be static
- Can be **public**, **private** or **protected**

#### 7.7 static Class Members

- static variables
  - Static variables are accessible through any object of the class
  - public static variables
    - Can also be accessed using scope resolution operator(::)

Employee::count

- private static variables
  - When no class member objects exist, can only be accessed via a **public static** member function
    - To call a public static member function combine the class name, the :: operator and the function name

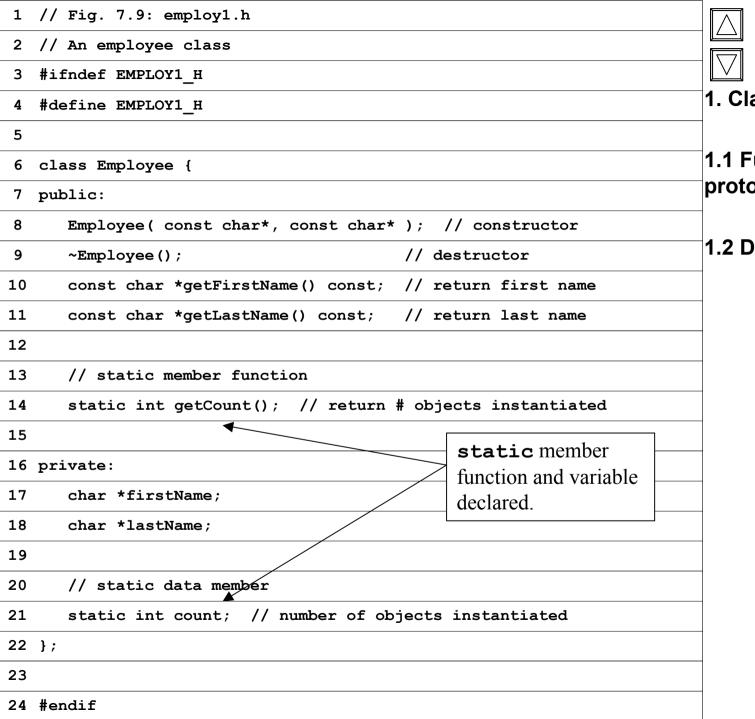
Employee::getCount()



### 7.7 static Class Members

### • Static functions

- static member functions cannot access non-static data or functions
- There is no **this** pointer for **static** functions, they exist independent of objects

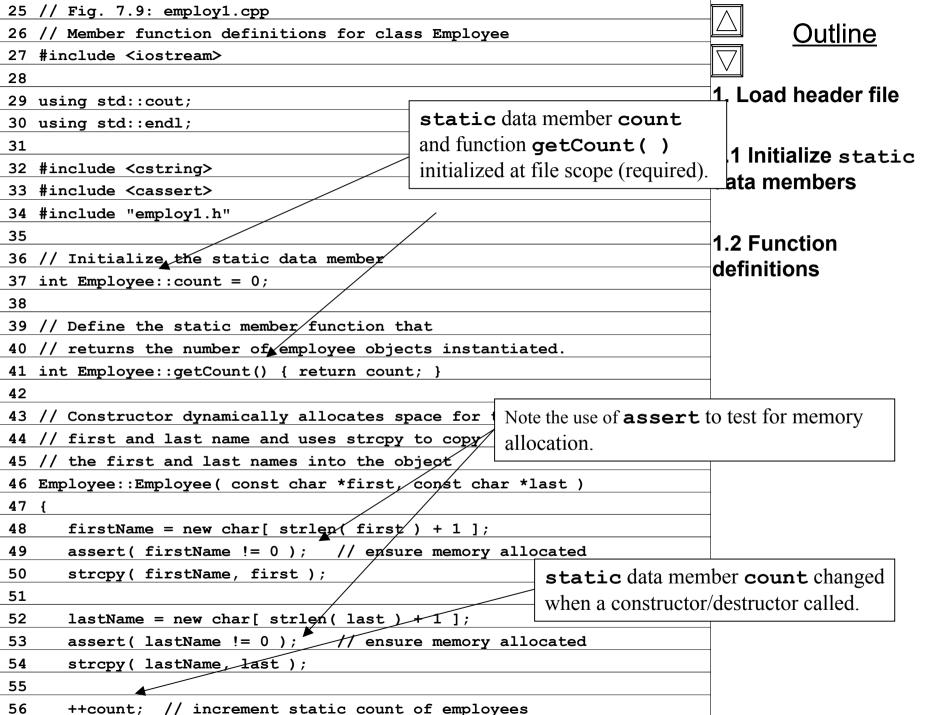


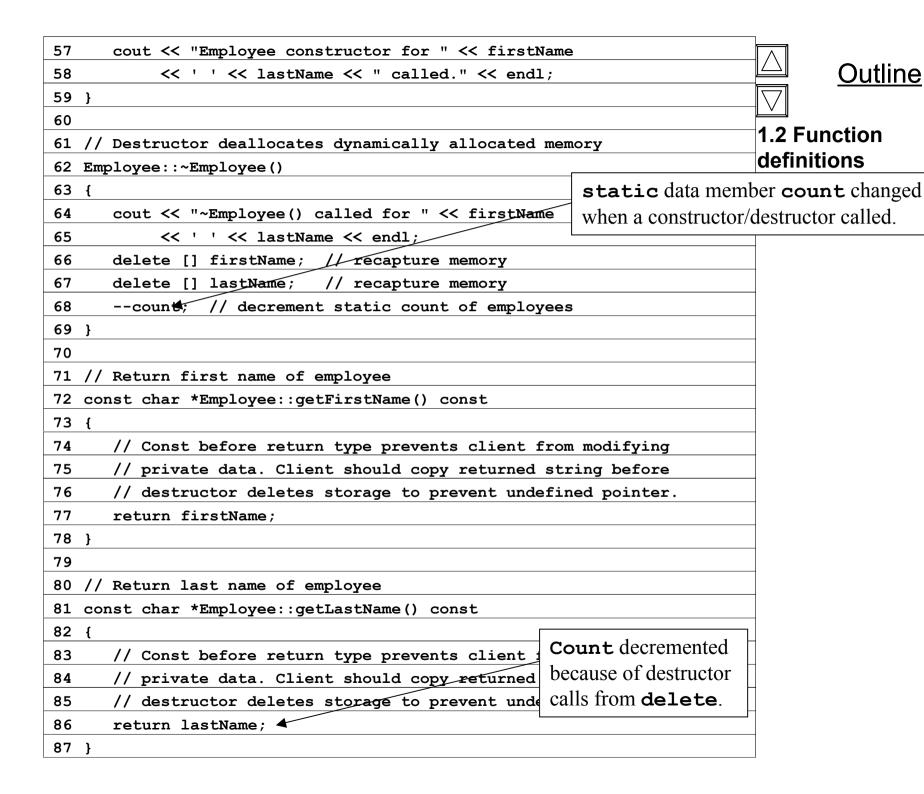
**Outline** 

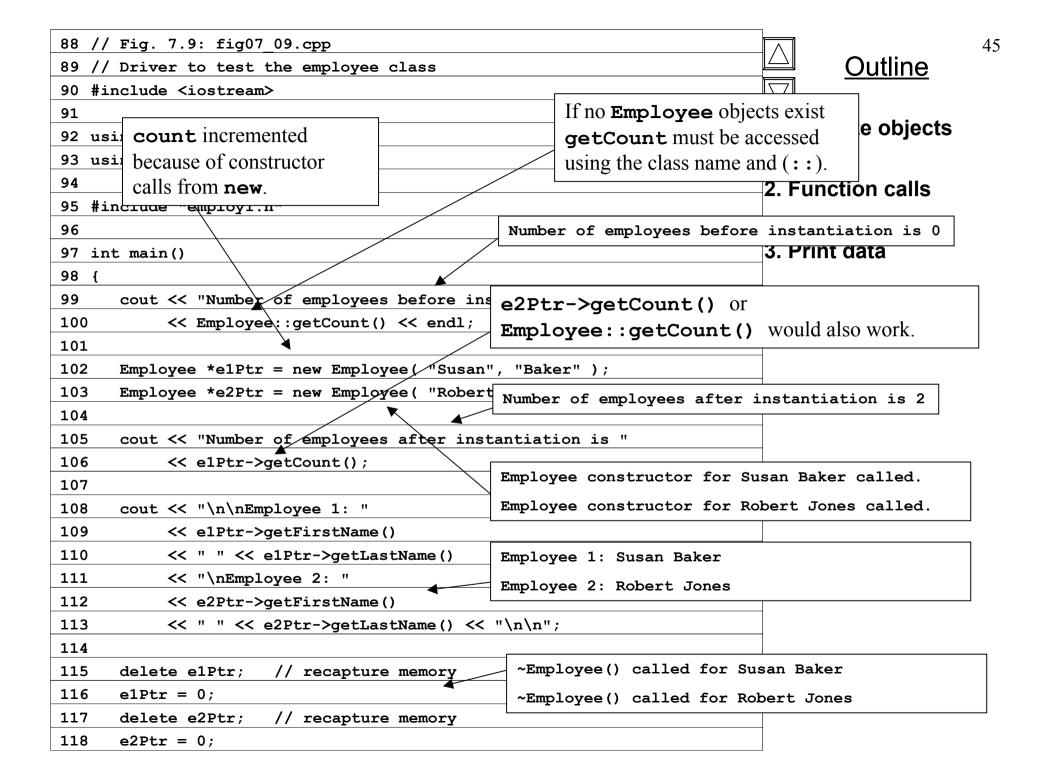
1. Class definition

1.1 Function prototypes

1.2 Declare variables







Outline

Outline

**Program Output** 

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  $\boldsymbol{0}$ 

# 7.8 Data Abstraction and Information Hiding

- Information hiding
  - Classes hide implementation details from clients
  - Example: stack data structure
    - Data elements added (pushed) onto the bottom and removed (popped) from top
    - Last-in, first-out (LIFO) data structure
    - Client does not care how stack is implemented, only wants LIFO data structure
- Abstract data types (ADTs)
  - Model real world objects
    - int, float are models for a numbers
- C++ is an extensible language
  - Standard data types cannot be changed, but new data types can be created

## 7.8.1 Example: Array Abstract Data Type

- Programmer can make an ADT array
  - Could include
    - Subscript range checking
    - An arbitrary range of subscripts instead of having to start with 0
    - Array assignment
    - Array comparison
    - Array input/output
    - Arrays that know their sizes
    - Arrays that expand dynamically to accommodate more elements

## 7.8.2 Example: String Abstract Data Type

- Strings in C++
  - C++ does not provide a built in string data type
    - Maximizes performance
  - Provides mechanisms for creating and implementing a string abstract data type
  - string class available in ANSI/ISO standard (Chapter 19)

# 7.8.3 Example: Queue Abstract Data Type

## • Queue

- Like waiting in line
  - FIFO First in, first out
- Enqueue
  - Put items in a queue one at a time, from the back
- Dequeue
  - Remove items from a queue one at a time, from the front
- Implementation hidden from clients

## Queue ADT

- Clients may not manipulate data structure directly
- Only queue member functions can access internal data

### 7.9 Container Classes and Iterators

- Container classes (collection classes)
  - Classes designed to hold collections of objects
  - Provide services such as insertion, deletion, searching, sorting, or testing an item
  - Examples:

Arrays, stacks, queues, trees and linked lists

- Iterator objects (iterators)
  - Object that returns the next item of a collection (or performs some action on the next item)
  - Can have several iterators per container
    - Book with multiple bookmarks
  - Each iterator maintains its own "position" information
  - Discussed further in chapter 20



## 7.10 Proxy Classes

## Proxy class

- Used to hide implementation details of a class
- Class that knows only the **public** interface of the class being hidden
- Enables clients to use class's services without giving access to class's implementation

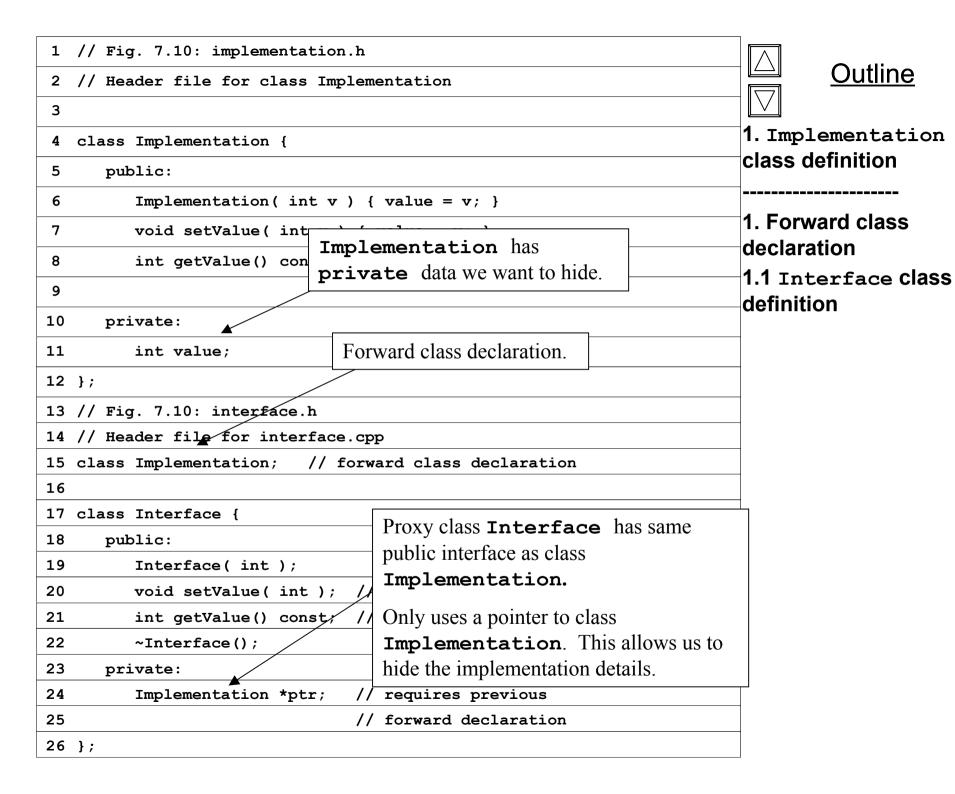
#### Forward class declaration

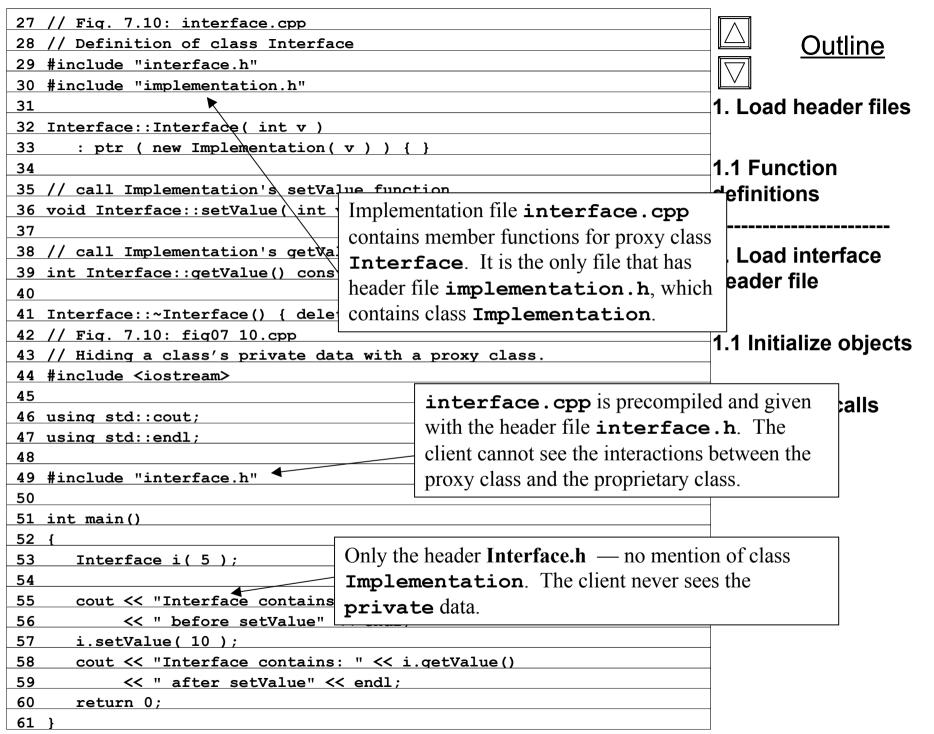
- Used when class definition only uses a pointer to another class
- Prevents the need for including the header file
- Declares a class before it is referenced
- Format:

class ClassToLoad;



Outline





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

**Outline** 

**Program Output**