Chapter 17 - C++ Classes: Part II

Outline
17.1 Introduction
17.2 `const` (Constant) Objects and `const` Member Functions
17.3 Composition: Objects as Members of Classes
17.4 `friend` Functions and `friend` Classes
17.5 Using the `this` Pointer
17.6 Dynamic Memory Allocation with Operators `new` and `delete`
17.7 `static` Class Members
17.8 Data Abstraction and Information Hiding
17.8.1 Example: Array Abstract Data Type
17.8.2 Example: String Abstract Data Type
17.8.3 Example: Queue Abstract Data Type
17.9 Container Classes and Iterators
Objectives

• In this chapter, you will learn:
  – To be able to create and destroy objects dynamically.
  – To be able to specify \texttt{const} (constant) objects and \texttt{const} member functions.
  – To understand the purpose of \texttt{friend} functions and \texttt{friend} classes.
  – To understand how to use \texttt{static} data members and member functions.
  – To understand the concept of a container class.
  – To understand the notion of iterator classes that walk through the elements of container classes.
  – To understand the use of the \texttt{this} pointer.
17.1 Introduction

- Chapters 16-18
  - Object-based programming

- Chapter 19-20
  - Polymorphism and inheritance
  - Object-oriented programming
17.2 \texttt{const} (Constant) Objects and \texttt{const} Member Functions

- **Principle of least privilege**
  - Only give objects permissions they need, no more

- **Keyword \texttt{const}**
  - Specify that an object is not modifiable
  - Any attempt to modify the object is a syntax error
  - For example:
    ```cpp
    const time noon(12, 0, 0);
    ```
  - Defines a \texttt{const} object \texttt{noon} of class \texttt{time} and initializes it to 12 noon
17.2 **const (Constant) Objects and **const** Member Functions (II)**

- **const** objects require **const** functions
  - Functions declared **const** cannot modify the object
  - **const** 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, and is appropriately declared **const**

- **Constructors / Destructors cannot be const**
  - They need to initialize variables (therefore modifying them)
// Fig. 17.1: time5.h

// Declaration of the class Time.
// Member functions defined in time5.cpp

#ifndef TIME5_H
#define TIME5_H

class Time {

public:

    Time( int = 0, int = 0, int = 0 ); // default constructor

    // set functions
    void setTime( int, int, int ); // set time
    void setHour( int );          // set hour
    void setMinute( int );        // set minute
    void setSecond( int );        // set second

    // get functions (normally declared const)
    int getHour() const;          // return hour
    int getMinute() const;        // return minute
    int getSecond() const;        // return second

    // print functions (normally declared const)
    void printMilitary() const;   // print military time
    void printStandard();         // print standard time

private:

};
```cpp
int hour;          // 0 - 23
int minute;       // 0 - 59
int second;       // 0 - 59
}; // end class Time

#include <iostream>

using std::cout;

#include "time5.h"

// Fig. 17.1: time5.cpp
// Member function definitions for Time class.
#include <iostream>

using std::cout;

#include "time5.h"

// Constructor function to initialize private data.
// Default values are 0 (see class definition).
Time::Time( int hr, int min, int sec )
    { setTime( hr, min, sec ); }

// Set the values of hour, minute, and second.
void Time::setTime( int h, int m, int s )
    {
        setHour( h );
        setMinute( m );
        setSecond( s );
    } // end function setTime
```

53 // Set the hour value
54 void Time::setHour( int h )
55 { hour = ( h >= 0 && h < 24 ) ? h : 0; }
56
57 // Set the minute value
58 void Time::setMinute( int m )
59 { minute = ( m >= 0 && m < 60 ) ? m : 0; }
60
61 // Set the second value
62 void Time::setSecond( int s )
63 { second = ( s >= 0 && s < 60 ) ? s : 0; }
64
65 // Get the hour value
66 int Time::getHour() const { return hour; }
67
68 // Get the minute value
69 int Time::getMinute() const { return minute; }
70
71 // Get the second value
72 int Time::getSecond() const { return second; }
// Display military format time: HH:MM
void Time::printMilitary() const
{
    cout << ( hour < 10 ? "0" : "" ) << hour << ":" << ( minute < 10 ? "0" : "" ) << minute;
} // end function printMilitary

// Display standard format time: HH:MM:SS AM (or PM)
void Time::printStandard()  // should be const
{
    cout << ( ( hour == 12 ) ? 12 : hour % 12 ) << ":" << ( minute < 10 ? "0" : "" ) << minute << ":" << ( second < 10 ? "0" : "" ) << second
    << ( hour < 12 ? " AM" : " PM" );
} // end function printStandard
# Fig. 17.1: fig17_01.cpp

// Attempting to access a const object with non-const member functions.

#include "time5.h"

int main()
{
    Time wakeUp(6, 45, 0); // non-constant object
    const Time noon(12, 0, 0); // constant object

    // MEMBER FUNCTION   OBJECT
    wakeUp.setHour(18); // non-const         non-const
    noon.setHour(12); // non-const         const

    wakeUp.getHour(); // const         non-const
    noon.getMinute(); // const         const
    noon.printMilitary(); // const         const
    noon.printStandard(); // non-const         const

    return 0;
} // end function main
Compiling...
Fig17_01.cpp

d:\fig17_01.cpp(14) : error C2662: 'setHour' : cannot convert 'this' pointer from 'const class Time' to 'class Time &'
Conversion loses qualifiers

d:\fig17_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)
17.2 const (Constant) Objects and const Member Functions (III)

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

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

- "increment( i )" initializes increment to the value of i.
- Any data member can be initialized using member initializer syntax
- consts and references must be initialized this way

- Multiple member initializers
  - Use comma-separated list after the colon
// Fig. 17.2: fig17_02.cpp
// Using a member initializer to initialize a constant of a built-in data type.

#include <iostream>

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

class Increment {
public:
  Increment(int c = 0, int i = 1);
  void addIncrement() { count += increment; }
  void print() const;

private:
  int count;
  const int increment; // const data member
}; // end class Increment

// Constructor for class Increment
Increment::Increment(int c, int i)
  : increment(i) // initializer for const member
    { count = c; }


// Print the data

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

int main()
{
    Increment value( 10, 5 );

    cout << "Before incrementing: ";
    value.print();

    for ( int j = 0; j < 3; j++ ) {
        value.addIncrement();
        cout << "After increment " << j + 1 << ": ";
        value.print();
    } // end for

    return 0;
} // end function 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
// Fig. 17.3: fig17_03.cpp
// Attempting to initialize a constant of
// a built-in data type with an assignment.
#include <iostream>
using std::cout;
using std::endl;

class Increment {
  public:
    Increment( int c = 0, int i = 1 );
    void addIncrement() { count += increment; }
    void print() const;
  private:
    int count;
    const int increment;
}; // end class Increment

// Constructor for class Increment
Increment::Increment( int c, int i )
{
    // Constant member 'increment' is not initialized
    count = c;
    increment = i; // ERROR: Cannot modify a const object
} // end Increment constructor
// Print the data

void Increment::print() const
{
    cout << "count = " << count << 
        "inc = " << increment << endl;
} // end function print

int main()
{
    Increment value(10, 5);
    cout << "Before incrementing: ";
    value.print();

    for ( int j = 0; j < 3; j++ ) {
        value.addIncrement();
        cout << "After increment " << j << 
            ": ";
        value.print();
    } // end for

    return 0;
} // end function main
Compiling...
Fig17_03.cpp
D:\Fig17_03.cpp(21) : error C2758: 'increment' : must be initialized in constructor base/member initializer list
D:\Fig17_03.cpp(16) : see declaration of 'increment'
D:\Fig17_03.cpp(23) : error C2166: l-value specifies const object
Error executing cl.exe.

test.exe - 2 error(s), 0 warning(s)
17.3 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)
  • Constructors called inside out
  • Destructors called outside in
17.3 Composition: Objects as Members of Classes (II)

- Example:

```cpp
Employee::Employee( char *fname, char *lname,
                     int bmonth, int bday, int byear,
                     int hmonth, int hday, int hyear )
  : birthDate( bmonth, bday, byear ),
    hireDate( hmonth, hday, hyear )
```

- Insert objects from Date class (birthDate and hireDate) into Employee class
- birthDate and hireDate have member initializers - they are probably consts in the Employee class
// Fig. 17.4: date1.h
// Declaration of the Date class.
// Member functions defined in date1.cpp

#ifndef DATE1_H
#define DATE1_H

class Date {

public:

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

private:

    int month; // 1-12
    int day; // 1-31 based on month
    int year; // any year

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

}; // end class Date

#endif
// Fig. 17.4: date1.cpp
// Member function definitions for Date class.
#include <iostream>

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

#include "date1.h"

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

    year = yr; // should validate yr
    day = checkDay( dy ); // validate the day
cout << "Date object constructor for date ";
print(); // interesting: a print with no arguments
    cout << endl;
} // end Date constructor

// Print Date object in form  month/day/year
void Date::print() const
    { cout << month << '/' << day << '/' << year; }

// Destructor: provided to confirm destruction order
Date::~Date()
    {
        cout << "Date object destructor for date ";
        print();
        cout << endl;
    } // end Date destructor

// Utility function to confirm proper day value
// based on month and year.
// Is the year 2000 a leap year?
int Date::checkDay( int testDay )
    {
        static const int daysPerMonth[13] =
            {0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};
if ( testDay > 0 && testDay <= daysPerMonth[ month ] )
    return testDay;

if ( month == 2 && // February: Check for leap year
    testDay == 29 &&
    ( year % 400 == 0 ||
    ( year % 4 == 0 && year % 100 != 0 ) ) )
    return testDay;

cout << "Day " << testDay << " invalid. Set to day 1.\n";

return 1; // leave object in consistent state if bad value
} // end function checkDay
// Fig. 17.4: emply1.h
// Declaration of the Employee class.
// Member functions defined in emply1.cpp

#ifndefEMPLY1_H
#defineEMPLY1_H

#include "date1.h"

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

private:
    char firstName[25];
    char lastName[25];
    const Date birthDate;
    const Date hireDate;
}; // end Employee constructor

#endif
// Fig. 17.4: emply1.cpp  
// Member function definitions for Employee class.  
#include <iostream>  
using std::cout;  
using std::endl;  

#include <cstring>  
#include "emply1.h"  
#include "date1.h"  

Employee::Employee( char *fname, char *lname,  
                    int bmonth, int bday, int byear,  
                    int hmonth, int hday, int hyear )  
  : birthDate( bmonth, bday, byear ),  
    hireDate( hmonth, hday, hyear )  
{
  // copy fname into firstName and be sure that it fits  
  int length = strlen( fname );  
  length = ( length < 25 ? length : 24 );  
  strncpy( firstName, fname, length );  
  firstName[ length ] = '\'0';
// copy lname into lastName and be sure that it fits
length = strlen(lname);
length = (length < 25 ? length : 24);
strncpy(lastName, lname, length);
lastName[length] = '\0';

  cout << "Employee object constructor: "
       << firstName << ' ' << lastName << endl;
} // end Employee constructor

void Employee::print() const
{
  cout << lastName << ', ' << firstName << "\nHired: ";
  hireDate.print();
  cout << " Birth date: ";
  birthDate.print();
  cout << endl;
} // end function print

// Destructor: provided to confirm destruction order
Employee::~Employee()
{
  cout << "Employee object destructor: "
       << lastName << ', ' << firstName << endl;
} // end Employee destructor
// Fig. 17.4: fig17_04.cpp
// Demonstrating composition: an object with member objects.
#include <iostream>
using std::cout;
using std::endl;
#include "emply1.h"

int main()
{
    Employee e("Bob", "Jones", 7, 24, 1949, 3, 12, 1988);
    cout << '\n';
    e.print();
    cout << "\nTest Date constructor with invalid values:\n";
    Date d( 14, 35, 1994 ); // invalid Date values
    cout << endl;
    return 0;
} // end function main
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
17.4 **friend Functions and friend Classes**

- **friend function and friend classes**
  - Can access private and protected (more later) members of another class
  - friend functions are not member functions of class
    - Defined outside of class scope

- **Properties**
  - 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)
17.4 friend Functions and friend Classes (II)

- friend declarations
  - friend function
    - Keyword friend before function prototype in class that is giving friendship.
    - friend int myFunction(int x);
    - Appears in the class granting friendship
  - friend class
    - Type friend class Classname in class granting friendship
    - If ClassOne granting friendship to ClassTwo,
      friend class ClassTwo;
      appears in ClassOne's definition
// Fig. 17.5: fig17_05.cpp
// Friends can access private members of a class.
#include <iostream>

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

// Modified Count class
class Count {
friend void setX( Count &, int ); // friend declaration
public:
    Count() { x = 0; } // constructor
    void print() const { cout << x << endl; } // output
private:
    int x; // data member
}; // end class Count

// Can modify private data of Count because
// setX is declared as a friend function of Count
void setX( Count &c, int val )
{
    c.x = val; // legal: setX is a friend of Count
} // end function setX
```cpp
int main()
{
    Count counter;
    cout << "counter.x after instantiation: ";
    counter.print();
    cout << "counter.x after call to setX friend function: ";
    setX( counter, 8 ); // set x with a friend
    counter.print();
    return 0;
} // end function main
```

**Program Output**

```
counter.x after instantiation: 0
counter.x after call to setX friend function: 8
```
// Fig. 17.6: fig17_06.cpp
// Non-friend/non-member functions cannot access
// private data of a class.
#include <iostream>

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

// Modified Count class
class Count {
    public:
    Count() { x = 0; }    // constructor
    void print() const { cout << x << endl; }    // output

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

// Function tries to modify private data of Count,
// but cannot because it is not a friend of Count.
void cannotSetX( Count &c, int val )
{
    c.x = val;    // ERROR: 'Count::x' is not accessible
}    // end function cannotSetX
25 int main()
26 {
27    Count counter;
28 
29    cannotSetX( counter, 3 ); // cannotSetX is not a friend
30    return 0;
31 } // end function main

Compiling...
Fig17_06.cpp
D:\Fig17_06.cpp(22) :
   error C2248: 'x' : cannot access private member declared in class 'Count'
   D:\Fig17_06.cpp(15) : see declaration of 'x'
   Error executing cl.exe.

test.exe - 1 error(s), 0 warning(s)
17.5 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

- **Example: class Employee**
  - For non-const member functions: `type Employee * const`
    - Constant pointer to an `Employee` object
  - For const member functions: `type const Employee * const`
    - Constant pointer to an constant `Employee` object
17.5 Using the this Pointer (II)

- Cascaded member function calls
  - Function returns a reference pointer to the same object
    ```cpp
    {return *this;}
    ```
  - Other functions can operate on that pointer
  - Functions that do not return references must be called last
17.5 Using the this Pointer (III)

- **Example**
  - 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)` and returns `*this` (reference to object), and 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 becomes
      ```
      t;
      ```
  - Has no effect
// Fig. 17.7: fig17_07.cpp
// Using the this pointer to refer to object members.
#include <iostream>

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

class Test {
public:
    Test( int = 0 );  // default constructor
    void print() const;

private:
    int x;
};  // end class Test

Test::Test( int a ) { x = a; }  // constructor

void Test::print() const  // ( ) around *this required
{
    cout << "        x = " << x
        << "\n    this->x = " << this->x
        << "\n    (*this).x = " << (*this).x << endl;
} // end function print

```cpp
int main()
{
    Test testObject( 12 );

    testObject.print();

    return 0;
} // end function main
```

```cpp
x = 12
this->x = 12
(*this).x = 12
```
// Fig. 17.8: time6.h
// Cascading member function calls.

// Declaration of class Time.
// Member functions defined in time6.cpp
#ifndef TIME6_H
#define TIME6_H

class Time {
  public:
    Time( int = 0, int = 0, int = 0 ); // default constructor

    // set functions
    Time &setTime( int, int, int ); // set hour, minute, second
    Time &setHour( int ); // set hour
    Time &setMinute( int ); // set minute
    Time &setSecond( int ); // set second

    // get functions (normally declared const)
    int getHour() const; // return hour
    int getMinute() const; // return minute
    int getSecond() const; // return second
};
#endif // TIME6_H
// print functions (normally declared const)
void printMilitary() const; // print military time
void printStandard() const; // print standard time

private:
int hour; // 0 - 23
int minute; // 0 - 59
int second; // 0 - 59
}; // end class Time

// Fig. 17.8: time6.cpp
// Member function definitions for Time class.
#include <iostream>

using std::cout;

#include "time6.h"

// Constructor function to initialize private data.
// Calls member function setTime to set variables.
// Default values are 0 (see class definition).
Time::Time( int hr, int min, int sec )
{ setTime( hr, min, sec ); }
// Set the values of hour, minute, and second.

Time &Time::setTime(int h, int m, int s)
{
    setHour(h);
    setMinute(m);
    setSecond(s);
    return *this; // enables cascading
} // end function setTime

// Set the hour value

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

// Set the minute value

Time &Time::setMinute(int m)
{
    minute = (m >= 0 && m < 60) ? m : 0;
    return *this; // enables cascading
} // end function setMinute
return *this; // enables cascading
}

// Get the hour value
int Time::getHour() const { return hour; }

// Get the minute value
int Time::getMinute() const { return minute; }

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

// Display military format time: HH:MM
void Time::printMilitary() const
{
    cout << ( hour < 10 ? "0" : "" ) << hour << ":
    << ( minute < 10 ? "0" : "" ) << minute;
}

// Display standard format time: HH:MM:SS AM (or PM)
void Time::printStandard() const
{
    cout << ( ( hour == 0 || hour == 12 ) ? 12 : hour % 12 )
    << ":" << ( minute < 10 ? "0" : "" ) << minute
    << ":" << ( second < 10 ? "0" : "" ) << second
    << ( hour < 12 ? " AM" : " PM" );
}
// Fig. 17.8: fig17_08.cpp
// Cascading member function calls together
// with the this pointer

#include <iostream>
using std::cout;
using std::endl;

#include "time6.h"

int main()
{
    Time t;

    t.setHour(18).setMinute(30).setSecond(22);
    cout << "Military time: ";
    t.printMilitary();
    cout << "
Standard time: ";
    t.printStandard();

    cout << "
New standard time: ";
    t.setTime(20, 20, 20).printStandard();
    cout << endl;

    return 0;
} // end function main
Military time: 18:30
Standard time: 6:30:22 PM

New standard time: 8:20:20 PM
17.6 Dynamic Memory Allocation with Operators
new and delete

- **new** and **delete**
  - Better dynamic memory allocation than C’s `malloc` and `free`
  - **new** - automatically creates object of proper size, calls constructor, returns pointer of the correct type
  - **delete** - destroys object and frees space

- **Example:**
  - `TypeName *typeNamePtr;`
    - Creates pointer to a `TypeName` object
  - `typeNamePtr = new TypeName;`
    - **new** creates `TypeName` object, returns pointer (which `typeNamePtr` is set equal to)
  - `delete typeNamePtr;`
    - Calls destructor for `TypeName` object and frees memory
17.6 Dynamic Memory Allocation with Operators

new and delete (II)

• Initializing objects

```cpp
double *thingPtr = new double( 3.14159 );
- Initializes object of type double to 3.14159

int *arrayPtr = new int[ 10 ];
- Create ten element int array, assign to arrayPtr.
- Use
  delete [] arrayPtr;
  to delete arrays
```
17.7 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
- Can be variables or functions
  - public, private, or protected
17.7 static Class Members (II)

- **Accessing static members**
  - **public static variables**: accessible through any object of the class
    - Or use class name and (::)
      Employee::count
  - **private static variables**: a **public static** member function must be used.
    - Prefix with class name and (::)
      Employee::getCount()
  - **static member functions** cannot access non-static data or functions
    - No this pointer, function exists independent of objects
// Fig. 17.9: employ1.h
// An employee class
#ifndef EMPLOY1_H
#define EMPLOY1_H

class Employee {
public:
    Employee( const char*, const char* ); // constructor
    ~Employee(); // destructor
    const char* getFirstName() const; // return first name
    const char* getLastName() const; // return last name

    // static member function
    static int getCount(); // return # objects instantiated

private:
    char* firstName;
    char* lastName;

    // static data member
    static int count; // number of objects instantiated
}; // end class Employee

#endif
// Fig. 17.9: employ1.cpp
// Member function definitions for class Employee
#include <iostream>

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

#include <cstring>
#include <cassert>
#include "employ1.h"

// Initialize the static data member
int Employee::count = 0;

// Define the static member function that
// returns the number of employee objects instantiated.
int Employee::getCount() { return count; }

// Constructor dynamically allocates space for the
// first and last name and uses strcpy to copy
// the first and last names into the object
Employee::Employee( const char *first, const char *last )
{
    firstName = new char[ strlen( first ) + 1 ];
    assert( firstName != 0 ); // ensure memory allocated
    strcpy( firstName, first );
lastName = new char[strlen(last) + 1];

assert(lastName != 0); // ensure memory allocated
strcpy(lastName, last);

++count; // increment static count of employees

cout << "Employee constructor for " << firstName << ' ' << lastName << " called." << endl;

Employee::~Employee()
{
    cout << "~Employee() called for " << firstName << ' ' << lastName << endl;
    delete[] firstName; // recapture memory
    delete[] lastName; // recapture memory
    --count; // decrement static count of employees
}

// Destructor deallocates dynamically allocated memory
const char *Employee::getLastName() const
{
    // Const before return type prevents client from modifying
    // private data. Client should copy returned string before
    // destructor deletes storage to prevent undefined pointer.
    return lastName;
} // end function getLastName

// Fig. 17.9: fig17_09.cpp
// Driver to test the employee class
#include <iostream>
#include "employ1.h"

int main()
{
    cout << "Number of employees before instantiation is "
        << Employee::getCount() << endl;  // use class name

    Employee *e1Ptr = new Employee("Susan", "Baker");
    Employee *e2Ptr = new Employee("Robert", "Jones");

    cout << "Number of employees after instantiation is "
        << e1Ptr->getCount();
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 0
17.8 Data Abstraction and Information Hiding

• Information hiding
  – Classes hide implementation details from clients
  – Example: stack data structure
    • Data elements like a pile of dishes - added (pushed) 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
17.8 Data Abstraction and Information Hiding (II)

- Abstract data types (ADTs)
  - Model real world objects
    - `int`, `float` are models for a number
    - Imperfect - finite size, precision, etc.

- C++ an extensible language
  - Base cannot be changed, but new data types can be created
17.8.1 Example: Array Abstract Data Type

• Array
  • Essentially a pointer and memory locations

• Programmer can make an ADT array
  • New capabilities
    • Subscript range checking, array assignment and comparison, dynamic arrays, arrays that know their sizes...

• New classes
  • Proprietary to an individual, to small groups or to companies, or placed in standard class libraries
Example: String Abstract Data Type

- C++ intentionally sparse
  - Reduce performance burdens
  - Use language to create what you need, i.e. a string class

- string not a built-in data type
  - Instead, C++ enables you to create your own string class
17.8.3 Example: Queue Abstract Data Type

- **Queue** - a waiting line
  - Used by computer systems internally
  - We need programs that simulate queues
- **Queue has well-understood behavior**
  - Enqueue - put things in a queue one at a time
  - Dequeue - get those things back one at a time on demand
  - Implementation hidden from clients
- **Queue ADT - stable internal data structure**
  - Clients may not manipulate data structure directly
  - Only queue member functions can access internal data
17.9 Container Classes and Iterators

- Container classes (collection classes)
  - Classes designed to hold collections of objects
    - 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 some action)
    - Can have several iterators per container
      - Book with multiple bookmarks
    - Each iterator maintains its own “position” information