9

Classes: A Deeper Look, Part 1

My object all sublime I shall achieve in time.

— W. S. Gilbert

Is it a world to hide virtues in?

— William Shakespeare

Don't be "consistent," but be simply true.

— Oliver Wendell Holmes, Jr.

This above all: to thine own self be true.

— William Shakespeare

OBJECTIVES

In this chapter you will learn:

- How to use a preprocessor wrapper to prevent multiple definition errors caused by including more than one copy of a header file in a source-code file.
- To understand class scope and accessing class members via the name of an object, a reference to an object or a pointer to an object.
- To define constructors with default arguments.
- How destructors are used to perform "termination housekeeping" on an object before it is destroyed.
- When constructors and destructors are called and the order in which they are called.
- The logic errors that may occur when a `public` member function of a class returns a reference to `private` data.
- To assign the data members of one object to those of another object by default memberwise assignment.

Outline

9.1 Introduction

- Integrated `Time` class case study
- Preprocessor wrapper
- Three types of "handles" on an object
  - Name of an object
  - Reference to an object
  - Pointer to an object
- Class functions
  - Predicate functions
  - Utility functions

9.1 Introduction (Cont.)

- Passing arguments to constructors
- Using default arguments in a constructor
- Destructor
  - Performs “termination housekeeping”

9.2 Time Class Case Study

- Preprocessor wrappers
  - Prevents code from being included more than once
    - `#ifndef` – "if not defined"
      - Skip this code if it has been included already
    - `#define`
      - Define a name so this code will not be included again
    - `#endif`
  - If the header has been included previously
    - Name is defined already and the header file is not included again
  - Prevents multiple-definition errors
  - Example
    -
```
#ifndef TIME_H
#define TIME_H
… // code
#endif
```

Outline

```cpp
// Fig. 9.1: Time.h
// Declaration of class Time.
// Member functions are defined in Time.cpp

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

// Time class definition
class Time
{
public:
   Time(); // constructor
   void setTime( int, int, int ); // set hour, minute and second
   void printUniversal(); // print time in universal-time format
   void printStandard(); // print time in standard-time format
private:
   int hour; // 0 - 23 (24-hour clock format)
   int minute; // 0 - 59
   int second; // 0 - 59
}; // end class Time

#endif
```

Preprocessor directive **`#ifndef`** determines whether a name is defined

Preprocessor directive **`#define`** defines a name (e.g., **`TIME_H`**)

Preprocessor directive **`#endif`** marks the end of the code that should not be included multiple times

Good Programming Practice 9.1

For clarity and readability, use each access specifier only once in a class definition. Place `public` members first, where they are easy to locate.

Software Engineering Observation 9.1

Each element of a class should have private visibility unless it can be proven that the element needs `public` visibility. This is another example of the principle of least privilege.

Error-Prevention Tip 9.1

Use `#ifndef`, `#define` and `#endif` preprocessor directives to form a preprocessor wrapper that prevents header files from being included more than once in a program.

Good Programming Practice 9.2

Use the name of the header file in upper case with the period replaced by an underscore in the `#ifndef` and `#define` preprocessor directives of a header file.

```
// Fig. 9.2: Time.cpp
// Member-function definitions for class Time.
#include <iostream>
using std::cout;

#include <iomanip>
using std::setfill;
using std::setw;

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

// Time constructor initializes each data member to zero.
// Ensures all Time objects start in a consistent state.
Time::Time()
{
   hour = minute = second = 0;
} // end Time constructor

// set new Time value using universal time; ensure that
// the data remains consistent by setting invalid values to zero
void Time::setTime( int h, int m, int s )
{
   hour = ( h >= 0 && h < 24 ) ? h : 0; // validate hour
   minute = ( m >= 0 && m < 60 ) ? m : 0; // validate minute
   second = ( s >= 0 && s < 60 ) ? s : 0; // validate second
} // end function setTime
```

Outline

Time.cpp

(1 of 2)

Ensure that **hour**, **minute** and **second** values remain valid

```
// print Time in universal-time format (HH:MM:SS)
void Time::printUniversal()
{
   cout << setfill( '0' ) << setw( 2 ) << hour << ":"
      << setw( 2 ) << minute << ":" << setw( 2 ) << second;
} // end function printUniversal

// print Time in standard-time format (HH:MM:SS AM or PM)
void Time::printStandard()
{
   cout << ( ( hour == 0 || hour == 12 ) ? 12 : hour % 12 ) << ":"
      << setfill( '0' ) << setw( 2 ) << minute << ":" << setw( 2 )
      << second << ( hour < 12 ? " AM" : " PM" );
} // end function printStandard
```

Using **`setfill`** stream manipulator to specify a fill character

`Time.cpp`

(2 of 2)

Outline

```
// Fig. 9.3: fig09_03.cpp
// Program to test class Time.
// NOTE: This file must be compiled with Time.cpp.
#include <iostream>
using std::cout;
using std::endl;

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

int main()
{
   Time t; // instantiate object t of class Time

   // output Time object t's initial values
   cout << "The initial universal time is ";
   t.printUniversal(); // 00:00:00
   cout << "\nThe initial standard time is ";
   t.printStandard(); // 12:00:00 AM

   t.setTime( 13, 27, 6 ); // change time

   // output Time object t's new values
   cout << "\n\nUniversal time after setTime is ";
   t.printUniversal(); // 13:27:06
   cout << "\nStandard time after setTime is ";
   t.printStandard(); // 1:27:06 PM

   t.setTime( 99, 99, 99 ); // attempt invalid settings
```

fig09_03.cpp

(1 of 2)

```
   // output t's values after specifying invalid values
   cout << "\n\nAfter attempting invalid settings:"
      << "\nUniversal time: ";
   t.printUniversal(); // 00:00:00
   cout << "\nStandard time: ";
   t.printStandard(); // 12:00:00 AM
   cout << endl;
   return 0;
} // end main
```

```
The initial universal time is 00:00:00
The initial standard time is 12:00:00 AM

Universal time after setTime is 13:27:06
Standard time after setTime is 1:27:06 PM

After attempting invalid settings:
Universal time: 00:00:00
Standard time: 12:00:00 AM
```

Common Programming Error 9.1

Attempting to initialize a non-`static` data member of a class explicitly in the class definition is a syntax error.

9.2 `Time` Class Case Study (Cont.)

- **Parameterized stream manipulator `setfill`**
  - **Specifies the fill character**
    - **Which is displayed when an output field wider than the number of digits in the output value**
    - **By default, fill characters appear to the left of the digits in the number**
  - **`setfill` is a "sticky" setting**
    - **Applies for all subsequent values that are displayed in fields wider than the value being displayed**

Error-Prevention Tip 9.2

Each sticky setting (such as a fill character or floating-point precision) should be restored to its previous setting when it is no longer needed. Failure to do so may result in incorrectly formatted output later in a program. Chapter 15, Stream Input/Output, discusses how to reset the fill character and precision.

9.2 `Time` Class Case Study (Cont.)

- Member function declared in a class definition but defined outside that class definition
  - Still within the class's scope
  - Known only to other members of the class unless referred to via
    - Object of the class
    - Reference to an object of the class
    - Pointer to an object of the class
    - Binary scope resolution operator
- Member function defined in the body of a class definition
  - C++ compiler attempts to inline calls to the member function

Performance Tip 9.1

Defining a member function inside the class definition inlines the member function (if the compiler chooses to do so). This can improve performance.

Software Engineering Observation 9.2

Defining a small member function inside the class definition does not promote the best software engineering, because clients of the class will be able to see the implementation of the function, and the client code must be recompiled if the function definition changes.

Software Engineering Observation 9.3

Only the simplest and most stable member functions (i.e., whose implementations are unlikely to change) should be defined in the class header.

Software Engineering Observation 9.4

Using an object-oriented programming approach can often simplify function calls by reducing the number of parameters to be passed. This benefit of object-oriented programming derives from the fact that encapsulating data members and member functions within an object gives the member functions the right to access the data members.

Software Engineering Observation 9.5

Member functions are usually shorter than functions in non-object-oriented programs, because the data stored in data members have ideally been validated by a constructor or by member functions that store new data. Because the data is already in the object, the member-function calls often have no arguments or at least have fewer arguments than typical function calls in non-object-oriented languages. Thus, the calls are shorter, the function definitions are shorter and the function prototypes are shorter. This facilitates many aspects of program development.

Error-Prevention Tip 9.3

The fact that member function calls generally take either no arguments or substantially fewer arguments than conventional function calls in non-object-oriented languages reduces the likelihood of passing the wrong arguments, the wrong types of arguments or the wrong number of arguments.

9.2 Time Class Case Study (Cont.)

- Using class `Time`
  - Once class Time has been defined, it can be used in declarations
    - `Time sunset;`
    - `Time arrayOfTimes[ 5 ];`
    - `Time &dinnerTime = sunset;`
    - `Time *timePtr = &dinnerTime;`

Performance Tip 9.2

Objects contain only data, so objects are much smaller than if they also contained member functions. Applying operator `sizeof` to a class name or to an object of that class will report only the size of the class's data members. The compiler creates one copy (only) of the member functions separate from all objects of the class. All objects of the class share this one copy. Each object, of course, needs its own copy of the class's data, because the data can vary among the objects. The function code is nonmodifiable (also called reentrant code or pure procedure) and, hence, can be shared among all objects of one class.

9.3 Class Scope and Accessing Class Members

- Class scope contains
  - Data members
    - Variables declared in the class definition
  - Member functions
    - Functions declared in the class definition
- Nonmember functions are defined at file scope

9.3 Class Scope and Accessing Class Members (Cont.)

- **Within a class's scope**
  - **Class members are accessible by all member functions**
- **Outside a class's scope**
  - `public` **class members are referenced through a handle**
    - **An object name**
    - **A reference to an object**
    - **A pointer to an object**

9.3 Class Scope and Accessing Class Members (Cont.)

- **Variables declared in a member function**
  - **Have block scope**
  - **Known only to that function**
- **Hiding a class-scope variable**
  - **In a member function, define a variable with the same name as a variable with class scope**
  - **Such a hidden variable can be accessed by preceding the name with the class name followed by the scope resolution operator (`::`)**

9.3 Class Scope and Accessing Class Members (Cont.)

- **Dot member selection operator (.)**
  - **Accesses the object's members**
  - **Used with an object's name or with a reference to an object**
- **Arrow member selection operator (->)**
  - **Accesses the object's members**
  - **Used with a pointer to an object**

```
// Fig. 9.4: fig09_04.cpp
// Demonstrating the class member access operators . and ->
#include <iostream>
using std::cout;
using std::endl;

// class Count definition
class Count
{
public: // public data is dangerous
   // sets the value of private data member x
   void setX( int value )
   {
      x = value;
   } // end function setX

   // prints the value of private data member x
   void print()
   {
      cout << x << endl;
   } // end function print

private:
   int x;
}; // end class Count
```

Outline

fig09_04.cpp

(1 of 2)

Outline

fig09_04.cpp

(2 of 2)

```
int main()
{
   Count counter; // create counter object
   Count *counterPtr = &counter; // create pointer to counter
   Count &counterRef = counter; // create reference to counter

   cout << "Set x to 1
   counter.setX( 1 ); // set data member x to 1
   counter.print(); // call member function print

   cout << "Set x to 2 and
   counterRef.setX( 2 ); // set data member x to 2
   counterRef.print(); // call member function print

   cout << "Set x to 3 and print using a pointer to an object: ";
   counterPtr->setX( 3 ); // set data member x to 3
   counterPtr->print(); // call member function print
   return 0;
} // end main
```

Using the dot member selection operator with an object

Using the dot member selection operator with a reference

Using the arrow member selection operator with a pointer

```
Set x to 1 and print using the object's name: 1
Set x to 2 and print using a reference to an object: 2
Set x to 3 and print using a pointer to an object: 3
```

9.4 Separating Interface from Implementation

- Separating a class definition and the class's member-function definitions
  - Makes it easier to modify programs
    - Changes in the class's implementation do not affect the client as long as the class's interface remains unchanged
  - Things are not quite this rosy
    - Header files do contain some portions of the implementation and hint about others
      - Inline functions need to be defined in header file
      - `private` members are listed in the class definition in the header file

Software Engineering Observation 9.6

Clients of a class do not need access to the class's source code in order to use the class. The clients do, however, need to be able to link to the class's object code (i.e., the compiled version of the class). This encourages independent software vendors (ISVs) to provide class libraries for sale or license. The ISVs provide in their products only the header files and the object modules. No proprietary information is revealed—as would be the case if source code were provided. The C++ user community benefits by having more ISV-produced class libraries available.

Software Engineering Observation 9.7

Information important to the interface to a class should be included in the header file. Information that will be used only internally in the class and will not be needed by clients of the class should be included in the unpublished source file. This is yet another example of the principle of least privilege.

9.5 Access Functions and Utility Functions

- **Access functions**
  - **Can read or display data**
  - **Can test the truth or falsity of conditions**
    - **Such functions are often called predicate functions**
    - **For example, `isEmpty` function for a class capable of holding many objects**
- **Utility functions (also called helper functions)**
  - **`private` member functions that support the operation of the class's `public` member functions**
  - **Not part of a class's `public` interface**
    - **Not intended to be used by clients of a class**

```
// Fig. 9.5: SalesPerson.h
// SalesPerson class definition.
// Member functions defined in SalesPerson.cpp.
#ifndef SALESP_H
#define SALESP_H

class SalesPerson
{
public:
   SalesPerson(); // constructor
   void getSalesFromUser(); // input sales from keyboard
   void setSales( int, double ); // set sales for a specific month
   void printAnnualSales(); // summarize and print sales
private:
   double totalAnnualSales(); // prototype for utility function
   double sales[ 12 ]; // 12 monthly sales figures
}; // end class SalesPerson

#endif
```

Prototype for a **`private`** utility function

Outline

```
// Fig. 9.6: SalesPerson.cpp
// Member functions for class SalesPerson.
#include <iostream>
using std::cout;
using std::cin;
using std::endl;
using std::fixed;

#include <iomanip>
using std::setprecision;

#include "SalesPerson.h" // include SalesPerson class definition

// initialize elements of array sales to 0.0
SalesPerson::SalesPerson()
{
   for ( int i = 0; i < 12; i++ )
      sales[ i ] = 0.0;
} // end SalesPerson constructor
```

SalesPerson.cpp

(1 of 3)

```cpp

// get 12 sales figures from the user at the keyboard
void SalesPerson::getSalesFromUser()
{
   double salesFigure;

   for ( int i = 1; i <= 12; i++ )
   {
      cout << "Enter sales amount for month " << i << ": ";
      cin >> salesFigure;
      setSales( i, salesFigure );
   } // end for
} // end function getSalesFromUser

// set one of the 12 monthly sales figures; function subtracts
// one from month value for proper subscript in sales array
void SalesPerson::setSales( int month, double amount )
{
   // test for valid month and amount values
   if ( month >= 1 && month <= 12 && amount > 0 )
      sales[ month - 1 ] = amount; // adjust for subscripts 0-11
   else // invalid month or amount value
      cout << "Invalid month or sales figure" << endl;
} // end function setSales
```


SalesPerson.cpp

(2 of 3)

Outline

```
// print total annual sales (with the help of utility function)
void SalesPerson::printAnnualSales()
{
   cout << setprecision( 2 ) << fixed
      << "\nThe total annual sales are: $"
      << totalAnnualSales() << endl; // call utility function
} // end function printAnnualSales

// private utility function to total annual sales
double SalesPerson::totalAnnualSales()
{
   double total = 0.0; // initialize total

   for ( int i = 0; i < 12; i++ ) // summarize sales results
      total += sales[ i ]; // add month i sales to total

   return total;
} // end function totalAnnualSales
```

Calling a `private` utility function

Definition of a `private` utility function

`SalesPerson.cpp`

```
// Fig. 9.7: fig09_07.cpp
// Demonstrating a utility function.
// Compile this program with SalesPerson.cpp

// include SalesPerson class definition from SalesPerson.h
#include "SalesPerson.h"

int main()
{
   SalesPerson s; // create SalesPerson object s

   s.getSalesFromUser(); // note simple sequential code;
   s.printAnnualSales(); // no control statements in main
   return 0;
} // end main
```

```
Enter sales amount for month 1: 5314.76
Enter sales amount for month 2: 4292.38
Enter sales amount for month 3: 4589.83
Enter sales amount for month 4: 5534.03
Enter sales amount for month 5: 4376.34
Enter sales amount for month 6: 5698.45
Enter sales amount for month 7: 4439.22
Enter sales amount for month 8: 5893.57
Enter sales amount for month 9: 4909.67
Enter sales amount for month 10: 5123.45
Enter sales amount for month 11: 4024.97
Enter sales amount for month 12: 5923.92

The total annual sales are: $60120.59
```

Software Engineering Observation 9.8

A phenomenon of object-oriented programming is that once a class is defined, creating and manipulating objects of that class often involve issuing only a simple sequence of member-function calls—few, if any, control statements are needed. By contrast, it is common to have control statements in the implementation of a class's member functions.

9.6 Time Class Case Study: Constructors with Default Arguments

- Constructors can specify default arguments
  - Can initialize data members to a consistent state
    - Even if no values are provided in a constructor call
  - Constructor that defaults all its arguments is also a default constructor
    - Can be invoked with no arguments
    - Maximum of one default constructor per class

```
// Fig. 9.8: Time.h
// Declaration of class Time.
// Member functions defined in Time.cpp.

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

// Time abstract data type definition
class Time
{
public:
   Time( int = 0, int = 0, int = 0 ); // default constructor

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

Prototype of a constructor with default arguments

Outline

```
   // get functions
   int getHour(); // return hour
   int getMinute(); // return minute
   int getSecond(); // return second

   void printUniversal(); // output time in universal-time format
   void printStandard(); // output time in standard-time format
private:
   int hour; // 0 - 23 (24-hour clock format)
   int minute; // 0 - 59
   int second; // 0 - 59
}; // end class Time

#endif
```

`Time.h`

(2 of 2)

Time.cpp


```cpp
// Fig. 9.9: Time.cpp
// Member-function definitions for class Time.
#include <iostream>
using std::cout;

#include <iomanip>
using std::setfill;
using std::setw;

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

// Time constructor initializes each data member to zero;
// ensures that Time objects start in a consistent state
Time::Time( int hr, int min, int sec )
{
   setTime( hr, min, sec ); // validate and set time
} // end Time constructor

// set new Time value using universal time; ensure that
// the data remains consistent by setting invalid values to zero
void Time::setTime( int h, int m, int s )
{
   setHour( h ); // set private field hour
   setMinute( m ); // set private field minute
   setSecond( s ); // set private field second
} // end function setTime
```

Parameters could receive the default values

Outline

```

// set hour value
void Time::setHour( int h )
{
   hour = ( h >= 0 && h < 24 ) ? h : 0; // validate hour
} // end function setHour

// set minute value
void Time::setMinute( int m )
{
   minute = ( m >= 0 && m < 60 ) ? m : 0; // validate minute
} // end function setMinute

// set second value
void Time::setSecond( int s )
{
   second = ( s >= 0 && s < 60 ) ? s : 0; // validate second
} // end function setSecond

// return hour value
int Time::getHour()
{
   return hour;
} // end function getHour

// return minute value
int Time::getMinute()
{
   return minute;
} // end function getMinute
```

`Time.cpp`

Outline

Time.cpp

(3 of 3)

```
// return second value
int Time::getSecond()
{
   return second;
} // end function getSecond

// print Time in universal-time format (HH:MM:SS)
void Time::printUniversal()
{
   cout << setfill( '0' ) << setw( 2 ) << getHour() << ":"
      << setw( 2 ) << getMinute() << ":" << setw( 2 ) << getSecond();
} // end function printUniversal

// print Time in standard-time format (HH:MM:SS AM or PM)
void Time::printStandard()
{
   cout << ( ( getHour() == 0 || getHour() == 12 ) ? 12 : getHour() % 12 )
      << ":" << setfill( '0' ) << setw( 2 ) << getMinute()
      << ":" << setw( 2 ) << getSecond() << ( hour < 12 ? " AM" : " PM" );
} // end function printStandard
```

Software Engineering Observation 9.9

If a member function of a class already provides all or part of the functionality required by a constructor (or other member function) of the class, call that member function from the constructor (or other member function). This simplifies the maintenance of the code and reduces the likelihood of an error if the implementation of the code is modified. As a general rule: Avoid repeating code.

Software Engineering Observation 9.10

Any change to the default argument values of a function requires the client code to be recompiled (to ensure that the program still functions correctly).

Outline

fig09_10.cpp

(1 of 3)

```
// Fig. 9.10: fig09_10.cpp
// Demonstrating a default constructor for class Time.
#include <iostream>
using std::cout;
using std::endl;

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

int main()
{
   Time t1; // all arguments defaulted
   Time t2( 2 ); // hour specified; minute and second defaulted
   Time t3( 21, 34 ); // hour and minute specified; second defaulted
   Time t4( 12, 25, 42 ); // hour, minute and second specified
   Time t5( 27, 74, 99 ); // all bad values specified

   cout << "Constructed with:\n\nt1: all arguments defaulted\n  ";
   t1.printUniversal(); // 00:00:00
   cout << "\n  ";
   t1.printStandard(); // 12:00:00 AM

   cout << "\n\nt2: hour specified; minute and second defaulted\n  ";
   t2.printUniversal(); // 02:00:00
   cout << "\n  ";
   t2.printStandard(); // 2:00:00 AM
```

Initializing **Time** objects using 0, 1, 2 and 3 arguments

Outline

```
   cout << "\n\nt3: hour and minute specified; second defaulted\n  ";
   t3.printUniversal(); // 21:34:00
   cout << "\n  ";
   t3.printStandard(); // 9:34:00 PM

   cout << "\n\nt4: hour, minute and second specified\n  ";
   t4.printUniversal(); // 12:25:42
   cout << "\n  ";
   t4.printStandard(); // 12:25:42 PM

   cout << "\n\nt5: all invalid values specified\n  ";
   t5.printUniversal(); // 00:00:00
   cout << "\n  ";
   t5.printStandard(); // 12:00:00 AM
   cout << endl;
   return 0;
} // end main
```

fig09_10.cpp

(2 of 3)

Outline

```
Constructed with:

t1: all arguments defaulted
  00:00:00
  12:00:00 AM

t2: hour specified; minute and second defaulted
  02:00:00
  2:00:00 AM

t3: hour and minute specified; second defaulted
  21:34:00
  9:34:00 PM

t4: hour, minute and second specified
  12:25:42
  12:25:42 PM

t5: all invalid values specified
  00:00:00
  12:00:00 AM
```

Invalid values passed to constructor, so object **`t5`** contains all default data

`fig09_10.cpp`

(3 of 3)

Common Programming Error 9.2

A constructor can call other member functions of the class, such as set or get functions, but because the constructor is initializing the object, the data members may not yet be in a consistent state. Using data members before they have been properly initialized can cause logic errors.

9.7 Destructors

- **Destructor**
  - **A special member function**
  - **Name is the tilde character (~) followed by the class name, e.g., `~Time`**
  - **Called implicitly when an object is destroyed**
    - **For example, this occurs as an automatic object is destroyed when program execution leaves the scope in which that object was instantiated**
  - **Does not actually release the object's memory**
    - **It performs termination housekeeping**
    - **Then the system reclaims the object's memory**
      - **So the memory may be reused to hold new objects**

9.7 Destructors (Cont.)

- Destructor (Cont.)
  - Receives no parameters and returns no value
    - May not specify a return type—not even `void`
  - A class may have only one destructor
    - Destructor overloading is not allowed
  - If the programmer does not explicitly provide a destructor, the compiler creates an "empty" destructor

Common Programming Error 9.3

It is a syntax error to attempt to pass arguments to a destructor, to specify a return type for a destructor (even `void` cannot be specified), to return values from a destructor or to overload a destructor.

Software Engineering Observation 9.11

As we will see in the remainder of the book, constructors and destructors have much greater prominence in C++ and object-oriented programming than is possible to convey after only our brief introduction here.

9.8 When Constructors and Destructors Are Called

- Constructors and destructors
  - Called implicitly by the compiler
    - Order of these function calls depends on the order in which execution enters and leaves the scopes where the objects are instantiated
  - Generally,
    - Destructor calls are made in the reverse order of the corresponding constructor calls
  - However,
    - Storage classes of objects can alter the order in which destructors are called

9.8 When Constructors and Destructors Are Called (Cont.)

- For objects defined in global scope
  - Constructors are called before any other function (including `main`) in that file begins execution
  - The corresponding destructors are called when `main` terminates
    - Function `exit`
      - Forces a program to terminate immediately
        - Does not execute the destructors of automatic objects
      - Often used to terminate a program when an error is detected
    - Function `abort`
      - Performs similarly to function `exit`
        - But forces the program to terminate immediately without allowing the destructors of any objects to be called
      - Usually used to indicate an abnormal termination of the program

9.8 When Constructors and Destructors Are Called (Cont.)

- For an automatic local object
  - Constructor is called when that object is defined
  - Corresponding destructor is called when execution leaves the object's scope
- For automatic objects
  - Constructors and destructors are called each time execution enters and leaves the scope of the object
  - Automatic object destructors are not called if the program terminates with an `exit` or `abort` function

9.8 When Constructors and Destructors Are Called (Cont.)

- For a `static` local object
  - Constructor is called only once
    - When execution first reaches where the object is defined
  - Destructor is called when `main` terminates or the program calls function `exit`
    - Destructor is not called if the program terminates with a call to function `abort`
- Global and `static` objects are destroyed in the reverse order of their creation

Outline

```
// Fig. 9.11: CreateAndDestroy.h
// Definition of class CreateAndDestroy.
// Member functions defined in CreateAndDestroy.cpp.
#include <string>
using std::string;

#ifndef CREATE_H
#define CREATE_H

class CreateAndDestroy
{
public:
   CreateAndDestroy( int, string ); // constructor
   ~CreateAndDestroy(); // destructor
private:
   int objectID; // ID number for object
   string message; // message describing object
}; // end class CreateAndDestroy

#endif
```

Prototype for destructor

`CreateAndDestroy.h`

(1 of 1)

```
// Fig. 9.12: CreateAndDestroy.cpp
// Member-function definitions for class CreateAndDestroy.
#include <iostream>
using std::cout;
using std::endl;

#include "CreateAndDestroy.h"// include CreateAndDestroy class definition

// constructor
CreateAndDestroy::CreateAndDestroy( int ID, string messageString )
{
   objectID = ID; // set object's ID number
   message = messageString; // set object's descriptive message

   cout << "Object " << objectID << "   constructor runs   "
      << message << endl;
} // end CreateAndDestroy constructor

// destructor
CreateAndDestroy::~CreateAndDestroy()
{
   // output newline for certain objects; helps readability
   cout << ( objectID == 1 || objectID == 6 ? "\n" : "" );

   cout << "Object " << objectID << "   destructor runs    "
      << message << endl;
} // end ~CreateAndDestroy destructor
```

Defining the class's destructor

CreateAndDestroy.cpp

(1 of 1)

Outline

Fig09_13.cpp

(1 of 3)

```
// Fig. 9.13: fig09_13.cpp
// Demonstrating the order in which constructors and
// destructors are called.
#include <iostream>
using std::cout;
using std::endl;

#include "CreateAndDestroy.h" // include CreateAndDestroy class definition

void create( void ); // prototype

CreateAndDestroy first( 1, "(global before main)" ); // global object

int main()
{
   cout << "\nMAIN FUNCTION: EXECUTION BEGINS" << endl;
   CreateAndDestroy second( 2, "(local automatic in main)" );
   static CreateAndDestroy third( 3, "(local static in main)" );

   create(); // call function to create objects

   cout << "\nMAIN FUNCTION: EXECUTION RESUMES" << endl;
   CreateAndDestroy fourth( 4, "(local automatic in main)" );
   cout << "\nMAIN FUNCTION: EXECUTION ENDS" << endl;
   return 0;
} // end main
```

Object created outside of **main**

Local automatic object created in **main**

Local **static** object created in **main**

Local automatic object created in **main**

```
// function to create objects
void create( void )
{
   cout << "\nCREATE FUNCTION: EXECUTION BEGINS" << endl;
   CreateAndDestroy fifth( 5, "(local automatic in create)" );
   static CreateAndDestroy sixth( 6, "(local st
   CreateAndDestroy seventh( 7, "(local automatic in create)" );
   cout << "\nCREATE FUNCTION: EXECUTION ENDS" << endl
} // end function create
```

Local automatic object created in **create**

Local **static** object created in **create**

Local automatic object created in **create**

Outline

```
Object 1   constructor runs   (global before main)

MAIN FUNCTION: EXECUTION BEGINS
Object 2   constructor runs   (local automatic in main)
Object 3   constructor runs   (local static in main)

CREATE FUNCTION: EXECUTION BEGINS
Object 5   constructor runs   (local automatic in create)
Object 6   constructor runs   (local static in create)
Object 7   constructor runs   (local automatic in create)

CREATE FUNCTION: EXECUTION ENDS
Object 7   destructor runs    (local automatic in create)
Object 5   destructor runs    (local automatic in create)

MAIN FUNCTION: EXECUTION RESUMES
Object 4   constructor runs   (local automatic in main)

MAIN FUNCTION: EXECUTION ENDS
Object 4   destructor runs    (local automatic in main)
Object 2   destructor runs    (local automatic in main)

Object 6   destructor runs    (local static in create)
Object 3   destructor runs    (local static in main)

Object 1   destructor runs    (global before main)
```

Fig09_13.cpp

(3 of 3)

9.9 `Time` Class Case Study: A Subtle Trap—Returning a Reference to a `private` Data Member

- Returning a reference to an object
  - Alias for the name of an object
    - An acceptable *lvalue* that can receive a value
      - May be used on the left side of an assignment statement
    - If a function returns a `const` reference
      - That reference cannot be used as a modifiable *lvalue*
  - One (dangerous) way to use this capability
    - A `public` member function of a class returns a reference to a `private` data member of that class
      - Client code could alter `private` data
      - Same problem would occur if a pointer to `private` data were returned

```
// Fig. 9.14: Time.h
// Declaration of class Time.
// Member functions defined in Time.cpp

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

class Time
{
public:
   Time( int = 0, int = 0, int = 0 );
   void setTime( int, int, int );
   int getHour();
   int &badSetHour( int ); // DANGEROUS reference return
private:
   int hour;
   int minute;
   int second;
}; // end class Time

#endif
```

Prototype for function that returns a reference

Outline

Time.h

(1 of 1)

```
// Fig. 9.15: Time.cpp
// Member-function definitions for Time class.
#include "Time.h" // include definition of class Time

// constructor function to initialize private data;
// calls member function setTime to set variables;
// default values are 0 (see class definition)
Time::Time( int hr, int min, int sec )
{
   setTime( hr, min, sec );
} // end Time constructor

// set values of hour, minute and second
void Time::setTime( int h, int m, int s )
{
   hour = ( h >= 0 && h < 24 ) ? h : 0; // validate hour
   minute = ( m >= 0 && m < 60 ) ? m : 0; // validate minute
   second = ( s >= 0 && s < 60 ) ? s : 0; // validate second
} // end function setTime
```


Time.cpp

Outline

Time.cpp

(2 of 2)

```
// return hour value
int Time::getHour()
{
   return hour;
} // end function getHour

// POOR PROGRAMMING PRACTICE:
// Returning a reference to a private data member.
int &Time::badSetHour( int hh )
{
   hour = ( hh >= 0 && hh < 24 ) ? hh : 0;
   return hour; // DANGEROUS reference return
} // end function badSetHour
```

Returning a reference to a **private** data member = DANGEROUS!

```
// Fig. 9.16: fig09_16.cpp
// Demonstrating a public member function that
// returns a reference to a private data member.
#include <iostream>
using std::cout;
using std::endl;

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

int main()
{
   Time t; // create Time object

   // initialize hourRef with the reference returned by badSetHour
   int &hourRef = t.badSetHour( 20 ); // 20 is a valid hour

   cout << "Valid hour before modification: " << hourRef;
   hourRef = 30; // use hourRef to set invalid value in Time object t
   cout << "\nInvalid hour after modification: " << t.getHour();
```

Modifying a **`private`** data member through a returned reference

Outline

Fig09_16.cpp

(1 of 2)

```
   // Dangerous: Function call that returns
   // a reference can be used as an lvalue!
   t.badSetHour( 12 ) = 74; // assign another invalid value to hour

   cout << "\n\n************                                            "
      << "POOR PROGRAMMING PR
      << "t.badSetHour( 12 )
      << t.getHour()
      << "\n*************************************************" << endl;
   return 0;
} // end main
```

Modifying **`private`** data by using a function call as an *lvalue*

```
Valid hour before modification: 20
Invalid hour after modification: 30

*************************************************
POOR PROGRAMMING PRACTICE!!!!!!!!
t.badSetHour( 12 ) as an lvalue, invalid hour: 74
*************************************************
```

Error-Prevention Tip 9.4

Returning a reference or a pointer to a `private` data member breaks the encapsulation of the class and makes the client code dependent on the representation of the class's data. So, returning pointers or references to `private` data is a dangerous practice that should be avoided.

9.10 Default Memberwise Assignment

- **Default memberwise assignment**
  - **Assignment operator (=)**
    - **Can be used to assign an object to another object of the same type**
      - **Each data member of the right object is assigned to the same data member in the left object**
    - **Can cause serious problems when data members contain pointers to dynamically allocated memory**

Outline

```
// Fig. 9.17: Date.h
// Declaration of class Date.
// Member functions are defined in Date.cpp

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

// class Date definition
class Date
{
public:
   Date( int = 1, int = 1, int = 2000 ); // default constructor
   void print();
private:
   int month;
   int day;
   int year;
}; // end class Date

#endif
```

Date.h

(1 of 1)

Outline

```
// Fig. 9.18: Date.cpp
// Member-function definitions for class Date.
#include <iostream>
using std::cout;
using std::endl;

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

// Date constructor (should do range checking)
Date::Date( int m, int d, int y )
{
   month = m;
   day = d;
   year = y;
} // end constructor Date

// print Date in the format mm/dd/yyyy
void Date::print()
{
   cout << month << '/' << day << '/' << year;
} // end function print
```

Date.cpp

(1 of 1)

Outline

```
// Fig. 9.19: fig09_19.cpp
// Demonstrating that class objects can be assigned
// to each other using default memberwise assignment.
#include <iostream>
using std::cout;
using std::endl;

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

int main()
{
   Date date1( 7, 4, 2004 );
   Date date2; // date2 defaults to 1/1/2000

   cout << "date1 = ";
   date1.print();
   cout << "\ndate2 = ";
   date2.print();

   date2 = date1; // default memberwise assignment

   cout << "\n\nAfter default memberwise assignment, date2 = ";
   date2.print();
   cout << endl;
   return 0;
} // end main
```

Memberwise assignment assigns data members of **`date1`** to **`date2`**

```
date1 = 7/4/2004
date2 = 1/1/2000

After default memberwise assignment, date2 = 7/4/2004
```

`date2` now stores the same date as **`date1`**

fig09_19.cpp

(1 of 1)

9.10 Default Memberwise Assignment (Cont.)

- **Copy constructor**
  - **Enables pass-by-value for objects**
    - **Used to copy original object's values into new object to be passed to a function or returned from a function**
  - **Compiler provides a default copy constructor**
    - **Copies each member of the original object into the corresponding member of the new object (i.e., memberwise assignment)**
  - **Also can cause serious problems when data members contain pointers to dynamically allocated memory**

Performance Tip 9.3

Passing an object by value is good from a security standpoint, because the called function has no access to the original object in the caller, but pass-by-value can degrade performance when making a copy of a large object. An object can be passed by reference by passing either a pointer or a reference to the object. Pass-by-reference offers good performance but is weaker from a security standpoint, because the called function is given access to the original object. Pass-by-`const`-reference is a safe, good-performing alternative (this can be implemented with a `const` reference parameter or with a pointer-to-`const`-data parameter).

9.11 Software Reusability

- **Many substantial class libraries exist and others are being developed worldwide**
- **Software is increasingly being constructed from existing, well-defined, carefully tested, well-documented, portable, high-performance, widely available components**
- **Rapid applications development (RAD)**
  - **Speeds the development of powerful, high-quality software through the mechanisms of reusable componentry**

9.11 Software Reusability (Cont.)

- Problems to solve before realizing the full potential of software reusability
  - Cataloging schemes
  - Licensing schemes
  - Protection mechanisms to ensure that master copies of classes are not corrupted
  - Description schemes so that designers of new systems can easily determine whether existing objects meet their needs
  - Browsing mechanisms to determine what classes are available and how closely they meet software developer requirements
  - Research and development problems
- Great motivation to solve these problems
  - Potential value of their solutions is enormous

9.12 (Optional) Software Engineering Case Study: Starting to Program the Classes of the ATM System

- **Visibility of an object's attributes and operations**
  - **Determined by access specifiers**
  - **Data members normally have `private` visibility**
  - **Member functions normally have `public` visibility**
  - **Utility functions normally have `private` visibility**
- **UML Visibility Markers**
  - **Placed before an operation or an attribute**
  - **Plus sign (+) indicates `public` visibility**
  - **Minus sign (–) indicates `private` visibility**

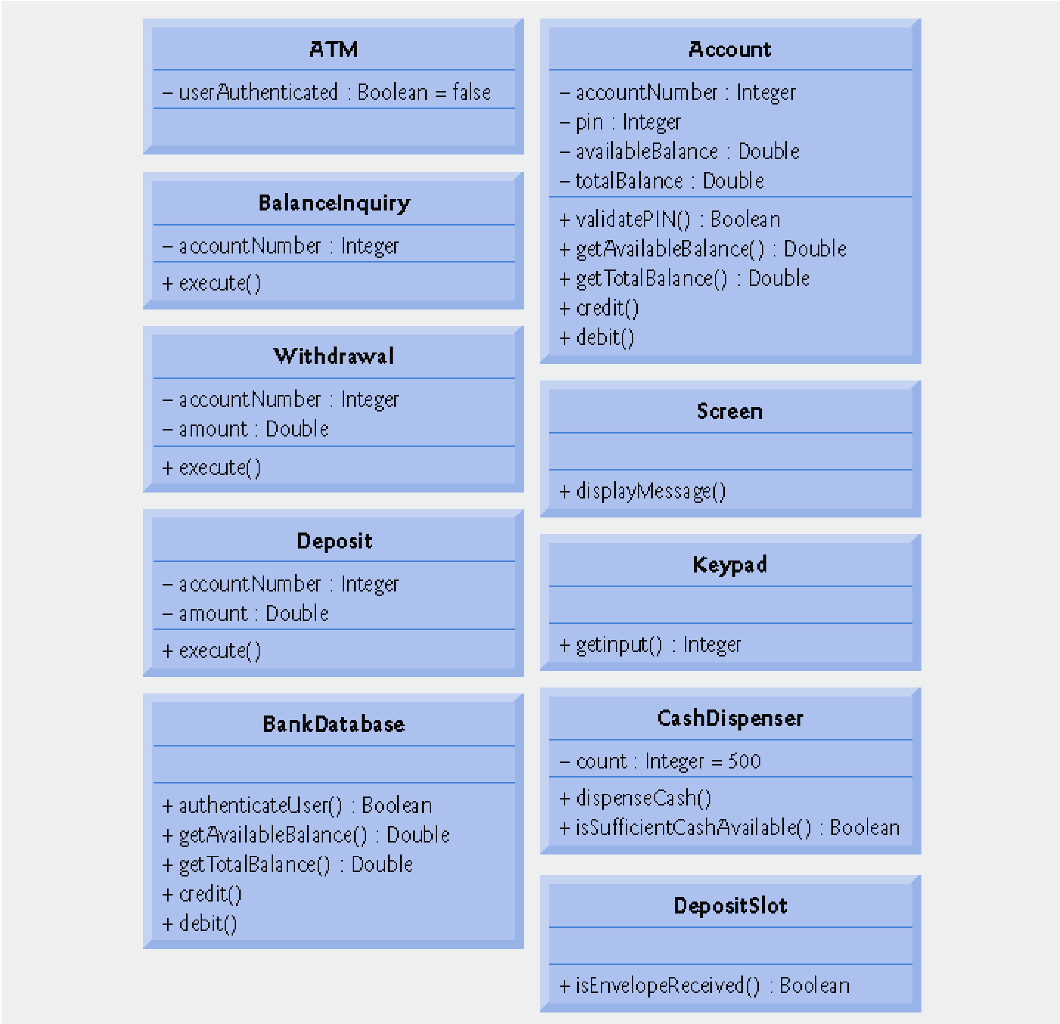

Fig. 9.20 | Class diagram with visibility markers.

9.12 (Optional) Software Engineering Case Study: Starting to Program the Classes of the ATM System (Cont.)

- **UML Navigability Arrows**
  - Arrows with stick arrowheads in a class diagram
  - Indicate in which direction an association can be traversed
  - Based on the collaborations modeled in communication and sequence diagrams
  - Help determine which objects need references or pointers to other objects
  - Bidirectional navigability
    - Indicated by arrows at both ends of an association or no arrows at all
    - Navigation can proceed in either direction across the association

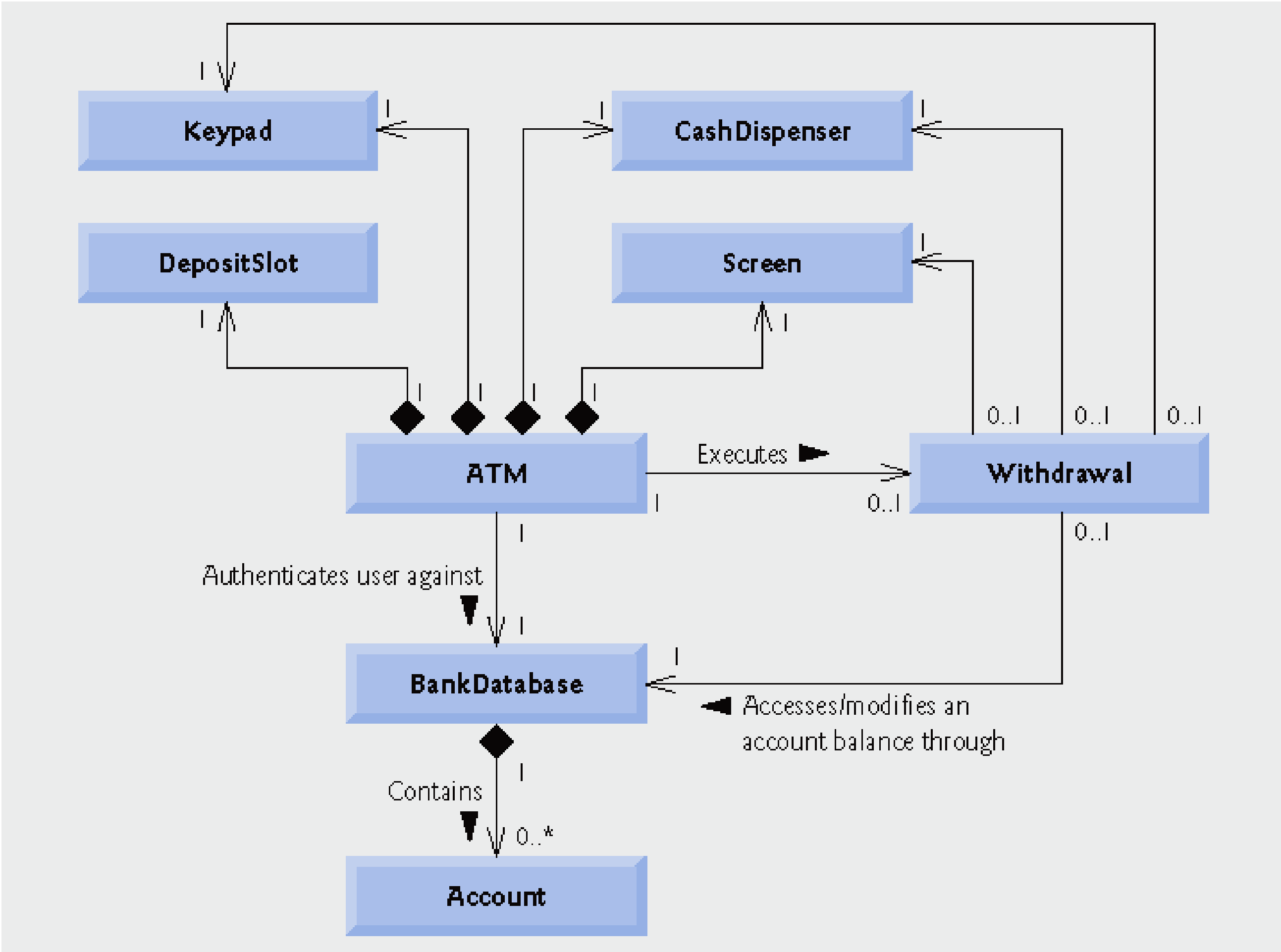

Fig. 9.21 | Class diagram with navigability arrows.

9.12 (Optional) Software Engineering Case Study: Starting to Program the Classes of the ATM System (Cont.)

- **Implementing a class from its UML design**
  - Use the name located in the first compartment of a class diagram to define the class in a header file
  - Use the attributes located in the class's second compartment to declare the data members
  - Use the associations described in the class diagram to declare references (or pointers, where appropriate) to other objects
    - Use forward declarations for references to classes (where possible) instead of including full header files
      - Helps avoid circular includes
        - Preprocessor problem that occurs when header file for class `A` `#includes` header file for class `B` and vice versa
  - Use the operations located in the class's third compartment to write the function prototypes of the class's member functions

```
// Fig. 9.22: Withdrawal.h
// Definition of class Withdrawal that represents a withdrawal transaction
#ifndef WITHDRAWAL_H
#define WITHDRAWAL_H

class Withdrawal
{
}; // end class Withdrawal

#endif // WITHDRAWAL_H
```

`#ifndef`, **`#define`** and **`#endif`** preprocessor directives help prevent multiple-definition errors

Class name is based on the top compartment of the class diagram

Outline

`fig09_22.cpp`

(1 of 1)

```
// Fig. 9.23: Withdrawal.h
// Definition of class Withdrawal that represents a withdrawal transaction
#ifndef WITHDRAWAL_H
#define WITHDRAWAL_H

class Withdrawal
{
private:
   // attributes
   int accountNumber; // account to withdraw funds from
   double amount; // amount to withdraw
}; // end class Withdrawal

#endif // WITHDRAWAL_H
```

Data members are based on the attributes in the middle compartment of the class diagram

```
// Fig. 9.24: Withdrawal.h
// Definition of class Withdra
#ifndef WITHDRAWAL_H
#define WITHDRAWAL_H

#include "Screen.h" // include definition of class Screen
#include "Keypad.h" // include definition of class Keypad
#include "CashDispenser.h" // include definition of class CashDispenser
#include "BankDatabase.h" // include definition of class BankDatabase

class Withdrawal
{
private:
   // attributes
   int accountNumber; // account to withdraw funds from
   double amount; // amount to withdraw

   // references to associated objects
   Screen &screen; // reference to ATM's screen
   Keypad &keypad; // reference to ATM's keypad
   CashDispenser &cashDispenser; // reference to ATM's cash dispenser
   BankDatabase &bankDatabase; // reference to the account info database
}; // end class Withdrawal

#endif // WITHDRAWAL_H
```

`#include` preprocessor directives for classes of associated objects

References are based on the associations in the class diagram

Withdrawal.h

(1 of 1)

Outline

Withdrawal.h

(1 of 1)

```
// Fig. 9.25: Withdrawal.h
// Definition of class Withdra                                          saction
#ifndef WITHDRAWAL_H
#define WITHDRAWAL_H

class Screen; // forward declaration of class Screen
class Keypad; // forward declaration of class Keypad
class CashDispenser; // forward declaration of class CashDispenser
class BankDatabase; // forward declaration of class BankDatabase

class Withdrawal
{
private:
   // attributes
   int accountNumber; // account to withdraw funds from
   double amount; // amount to withdraw

   // references to associated objects
   Screen &screen; // reference to ATM’s screen
   Keypad &keypad; // reference to ATM's keypad
   CashDispenser &cashDispenser; // reference to ATM's cash dispenser
   BankDatabase &bankDatabase; // reference to the account info database
}; // end class Withdrawal

#endif // WITHDRAWAL_H
```

Forward declarations of classes for which this class has references

Outline

Withdrawal.h

(1 of 1)

```
// Fig. 9.26: Withdrawal.h
// Definition of class Withdrawal that represents a withdrawal transaction
#ifndef WITHDRAWAL_H
#define WITHDRAWAL_H

class Screen; // forward declaration of class Screen
class Keypad; // forward declaration of class Keypad
class CashDispenser; // forward declaration of class CashDispenser
class BankDatabase; // forward declaration of class BankDatabase

class Withdrawal
{
public:
   // operations
   void execute(); // perform the transaction
private:
   // attributes
   int accountNumber; // account to withdraw funds from
   double amount; // amount to withdraw

   // references to associated objects
   Screen &screen; // reference to ATM’s screen
   Keypad &keypad; // reference to ATM's keypad
   CashDispenser &cashDispenser; // reference to ATM's cash dispenser
   BankDatabase &bankDatabase; // reference to the account info database
}; // end class Withdrawal

#endif // WITHDRAWAL_H
```

Member functions are based on operations in the bottom compartment of the class diagram

Software Engineering Observation 9.12

Several UML modeling tools can convert UML-based designs into C++ code, considerably speeding the implementation process. For more information on these "automatic" code generators, refer to the Internet and Web resources listed at the end of Section 2.8.

```cpp
// Fig. 9.27: Account.h
// Account class definition. Represents a bank account.
#ifndef ACCOUNT_H
#define ACCOUNT_H

class Account
{
public:
   bool validatePIN( int ); // is user-specified PIN correct?
   double getAvailableBalance(); // returns available balance
   double getTotalBalance(); // returns total balance
   void credit( double ); // adds an amount to the Account
   void debit( double ); // subtracts an amount from the Account
private:
   int accountNumber; // account number
   int pin; // PIN for authentication
   double availableBalance; // funds available for withdrawal
   double totalBalance; // funds available + funds waiting to clear
}; // end class Account

#endif // ACCOUNT_H
```


Account.h