Chapter 6: Classes and Data Abstraction

Outline

6.1 Introduction
6.2 Structure Definitions
6.3 Accessing Members of Structures
6.4 Implementing a User-Defined Type Time with a Struct
6.5 Implementing a Time Abstract Data Type with a Class
6.6 Class Scope and Accessing Class Members
6.7 Separating Interface from Implementation
6.8 Controlling Access to Members
6.9 Access Functions and Utility Functions
6.10 Initializing Class Objects: Constructors
6.11 Using Default Arguments with Constructors
6.12 Using Destructors
6.13 When Constructors and Destructors Are Called
6.14 Using Data Members and Member Functions
6.15 A Subtle Trap: Returning a Reference to a Private Data Member
6.16 Assignment by Default Memberwise Copy
6.17 Software Reusability

6.1 Introduction

- Object-oriented programming (OOP)
  - Encapsulates data (attributes) and functions (behavior) into packages called classes
- Information hiding
  - Implementation details are hidden within the classes themselves
- Classes
  - Classes are the standard unit of programming
  - A class is like a blueprint – reusable
  - Objects are instantiated (created) from the class
  - For example, a house is an instance of a “blueprint class”

6.2 Structure Definitions

- Structures
  - Aggregate data types built using elements of other types
  ```
  struct Time {
    int hour;
    int minute;
    int second;
  };
  ```
  - Members of the same structure must have unique names
  - Two different structures may contain members of the same name
  - Each structure definition must end with a semicolon

6.3 Accessing Members of Structures

- Member access operators:
  - Dot operator (.) for structures and objects
  - Arrow operator (->) for pointers
  ```
  cout << timeObject.hour;  // Print member hour of timeObject.
  OR
  timePtr->hour is the same as (*timePtr).hour
  ```
  - Parentheses required: * has lower precedence than .

6.4 Implementing a User-Defined Type Time with a Struct

6.5 Implementing a Time Abstract Data Type with a Class

6.6 Class Scope and Accessing Class Members

6.7 Separating Interface from Implementation

6.8 Controlling Access to Members

6.9 Access Functions and Utility Functions

6.10 Initializing Class Objects: Constructors

6.11 Using Default Arguments with Constructors

6.12 Using Destructors

6.13 When Constructors and Destructors Are Called

6.14 Using Data Members and Member Functions

6.15 A Subtle Trap: Returning a Reference to a Private Data Member

6.16 Assignment by Default Memberwise Copy

6.17 Software Reusability

// Fig. 6.1: fig06_01.cpp  // Create a structure, set its members, and print it.
#include <iostream>

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

struct Time {    // structure definition
    int hour;     // 0-23
    int minute;   // 0-59
    int second;   // 0-59
};

void printMilitary( const Time & );  // prototype
void printStandard( const Time & );  // prototype

int main()
{
    Time dinnerTime;    // variable of new type Time

    // set members to valid values
    dinnerTime.hour = 18;
    dinnerTime.minute = 30;
    dinnerTime.second = 0;

    cout << "Dinner will be held at " <<
    printMilitary( dinnerTime ) << " military time,
which is " <<
    printStandard( dinnerTime ) << " standard time."
};

// Fig. 6.1: fig06_01.cpp  // Create a structure, set its members, and print it.
#include <iostream>

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

struct Time {    // structure definition
    int hour;     // 0-23
    int minute;   // 0-59
    int second;   // 0-59
};

void printMilitary( const Time & );  // prototype
void printStandard( const Time & );  // prototype

int main()
{
    Time dinnerTime;    // variable of new type Time

    // set members to valid values
    dinnerTime.hour = 18;
    dinnerTime.minute = 30;
    dinnerTime.second = 0;

    cout << "Dinner will be held at " <<
    printMilitary( dinnerTime ) << " military time,
which is " <<
    printStandard( dinnerTime ) << " standard time."
};
6.5 Implementing a Time Abstract Data Type with a Class

• Classes
  – Model objects that have attributes (data members) and behaviors (member functions)
  – Defined using keyword class
  – Have a body delineated with braces ({ and })
  – Class definitions terminate with a semicolon

class Time
{
  public:
    // Constructor
    Time();

    // Time constructor initializes each data member to zero.
    Time( int hour, int minute, int second );

    // Set a new Time value using military time. Performs validity checking on time.
    void setTime( int h, int m, int s );

    bool isValidMilitary( const Time &t );

    // Set the time to an invalid hour, then print it.
    void setInvalidHour( const Time &t );

    // Print the time in military format
    void printMilitary();

    // Print the time in standard format
    void printStandard();

  private:
    int hour, minute, second; // 0-23, 0-59, 0-59

  public:
    int getHour() const { return hour; }
    int getMinute() const { return minute; }
    int getSecond() const { return second; }

    bool operator==( const Time &t ) const { return hour == t.hour && minute == t.minute && second == t.second; }

    // Equality operator
    bool operator!=( const Time &t ) const { return ! (*this == t); }

    // Static member function that prints the current date and time in standard format
    static void printStandard();

    // Static member function that prints the current date and time in military format
    static void printMilitary();
};

6.5 Implementing a Time Abstract Data Type with a Class

• Member access specifiers
  – Classes can limit the access to their member functions and data
  – The three types of access a class can grant are:
    • Public — Accessible wherever the program has access to an object of the class
    • Private — Accessible only to member functions of the class
    • Protected — Similar to private and discussed later

• Constructor
  – Special member function that initializes the data members of a class object
  – Cannot return values
  – Have the same name as the class

7. Define the functions printMilitary and printStandard

8. Print the time in military format

9. Print the time in standard format

10. Print the time with invalid values

11. Define a Time class

12. Define default values for the time

© 2000 Prentice Hall, Inc. All rights reserved.
Outline

3. Outline

13. Outline

33. // Print Time in military format
34. void Time::printMilitary()
35. {
36.    cout << ( hour < 10 ? "0" : "" ) << hour << ":
37.        ( minute < 10 ? "0" : "" ) << minute;
38. }
39.
40. // Print Time in standard format
41. void Time::printStandard()
42. {
43.    cout << ( ( hour == 0 || hour == 12 ) ? 12 : hour % 12 )
44.         << ":" << ( minute < 10 ? "0" : "" ) << minute
45.         << ":" << ( second < 10 ? "0" : "" ) << second
46.         << ( hour < 12 ? " AM" : " PM" );
47. }
48.
49. // Driver to test simple class Time
50. int main()
51. {
52.    Time t;  // instantiate object t of class Time
53. 
54.    cout << "The initial military time is ";
55.    t.printMilitary();
56.    cout << "The initial standard time is ";
57.    t.printStandard();
58. 
59.    t.setTime( 13, 27, 6 );
60.    cout << "Military time after setTime is ";
61.    t.printMilitary();
62.    cout << "Standard time after setTime is ";
63.    t.printStandard();
64. 
65.    t.setTime( 99, 99, 99 );  // attempt invalid settings
66.    cout << "After attempting invalid settings:
67.         Military time: ";
68.    t.printMilitary();
69.    cout << "Standard time: ";
70.    t.printStandard();
71.    cout << endl;
72.    return 0;
73. }
74.

6.5 Implementing a Time Abstract Data Type with a Class

• Destructors
  – Functions with the same name as the class but preceded with a tilde character (~)
  – Cannot take arguments and cannot be overloaded
  – Performs "termination bookkeeping"
• Binary scope resolution operator (::)
  – Combines the class name with the member function name
• Different classes can have member functions with the same name
• Format for defining member functions

  ReturnType ClassName::MemberFunctionName( ){...

6.6 Class Scope and Accessing Class Members

• Class scope
  – Data members and member functions
• File scope
  – Non member functions
• Inside a scope
  – Members accessible by all member functions
    – Referenced by name
• Outside a scope
  – Members are referenced through handles
    – An object name, a reference to an object or a pointer to an object

6.5 Implementing a Time Abstract Data Type with a Class

• If a member function is defined inside the class
  – Scope resolution operator and class name are not needed
  – Defining a function outside a class does not change it being public/private
• Classes encourage software reuse
  – Inheritance allows new classes to be derived from old ones
6.7 Separating Interface from Implementation

- Separating interface from implementation
  - Makes it easier to modify programs
  - Header files
    - Contains class definitions and function prototypes
  - Source-code files
    - Contains member function definitions
### 6.9 Access Functions and Utility Functions

- **Utility functions**
  - `private` functions that support the operation of public functions
  - Not intended to be used directly by clients

- **Access functions**
  - `public` functions that read/display data or check conditions
  - Allow `public` functions to check `private` data

- **Following example**
  - Program to take in monthly sales and output the total
  - Implementation not shown, only access functions

### 6.10 Initializing Class Objects: Constructors

- **Constructors**
  - Initialize class members
  - Same name as the class
  - No return type
  - Member variables can be initialized by the constructor or set afterwards

- **Passing arguments to a constructor**
  - When an object of a class is declared, initializers can be provided
  - Format of declaration with initializers
    - `Class-type ObjectName(value1, value2,...);`
  - Default arguments may also be specified in the constructor prototype

---

**Program Output**

```
Enter sales amount for month 1: 5923.92
Enter sales amount for month 11: 4024.97
Enter sales amount for month 10: 5123.45
Enter sales amount for month 8: 5893.57
Enter sales amount for month 4: 5534.03
Enter sales amount for month 3: 4589.83

The total annual sales are: $60120.59
```
6.12 Using Destructors

- **Destructors**
  - Are member function of class
  - Perform termination housekeeping before the system
    reclaims the object’s memory
  - Complement of the constructor
  - Name is tilde (~) followed by the class name (i.e., \~Time)
    - Recall that the constructor’s name is the class name
    - Receives no parameters, returns no value
    - One destructor per class
    - No overloading allowed

6.13 When Constructors and Destructors Are Called

- **Constructors and destructors called automatically**
  - Order depends on scope of objects
- **Global scope objects**
  - Constructors called before any other function (including main)
  - Destructors called when main terminates (or exit function called)
  - Destructors not called if program terminates with abort
- **Automatic local objects**
  - Constructors called when objects are defined
  - Destructors called when objects leave scope
    - i.e., when the block in which they are defined is exited
  - Destructors not called if the program ends with exit or abort
6.14 Using Data Members and Member Functions

- Member functions
  - Allow clients of the class to set (i.e., write) or get (i.e., read) the values of private data members
  - Example:
    - Adjusting a customer’s bank balance
    - `private` data member `balance` of a class `BankAccount`
      - could be modified through the use of member function `computeInterest`
    - A member function that sets data member `interestRate` could be called `setInterestRate` and a member function that returns the `interestRate` could be called `getInterestRate`
      - Providing `set` and `get` functions does not make private variables `public`
      - A set function should ensure that the new value is valid

6.15 A Subtle Trap: Returning a Reference to a Private Data Member

- Reference to an object
  - Alias for the name of the object
  - May be used on the left side of an assignment statement
  - Reference can receive a value, which changes the original object as well
- Returning references
  - `public` member functions can return non-`const` references to `private` data members
    - Should be avoided, breaks encapsulation

6.16 Assignment by Default Memberwise Copy

- Assigning objects
  - An object can be assigned to another object of the same type using the assignment operator (=)
  - Member by member copy

- Objects may be
  - Passed as function arguments
  - Returned from functions (call-by-value default)

6.17 Software Reusability

- Software reusability
  - Implementation of useful classes
  - Class libraries exist to promote reusability
    - Allows for construction of programs from existing, well-defined, carefully tested, well-documented, portable, widely available components
    - Speeds development of powerful, high-quality software