Chapter 6: Classes and Data Abstraction

Outline

6.1 Introduction

- Object-oriented programming (OOP)
  - Encapsulates data (attributes) and functions (behavior) into packages called classes
- Information hiding
  - Class objects communicate across well-defined interfaces
  - Implementation details hidden within classes themselves
- User-defined (programmer-defined) types: classes
  - Data (data members)
  - Functions (member functions or methods)
  - Similar to blueprints – reusable
  - Class instance: object

6.2 Structure Definitions

- Structures
  - Aggregate data types built using elements of other types

```
struct Time {
     int hour;
     int minute;
     int second;
 };
```

Structure tag

Structure members

- Structure member naming
  - In same **`struct`**: must have unique names
  - In different **`struct`**s: can share name
- **`struct`** definition must end with semicolon

6.2 Structure Definitions

- Self-referential structure
  - Structure member cannot be instance of enclosing **`struct`**
  - Structure member can be pointer to instance of enclosing **`struct`** (self-referential structure)
    - Used for linked lists, queues, stacks and trees
- **`struct`** definition
  - Creates new data type used to declare variables
  - Structure variables declared like variables of other types
  - Examples:
    - `Time timeObject;`
    - `Time timeArray[ 10 ];`
    - `Time *timePtr;`
    - `Time &timeRef = timeObject;`

6.3 Accessing Structure Members

- Member access operators
  - Dot operator (`.`) for structure and class members
  - Arrow operator (`->`) for structure and class members via pointer to object
  - Print member **`hour`** of **`timeObject`**:

```
cout << timeObject.hour;
```

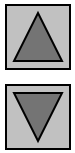
OR

```
timePtr = &timeObject;
cout << timePtr->hour;
```

  - **`timePtr->hour`** same as **`( *timePtr ).hour`**
    - Parentheses required
      - `*` lower precedence than `.`

6.4 Implementing a User-Defined Type `Time` with a `struct`

- Default: structures passed by value
  - Pass structure by reference
    - Avoid overhead of copying structure
- C-style structures
  - No “interface”
    - If implementation changes, all programs using that **`struct`** must change accordingly
  - Cannot print as unit
    - Must print/format member by member
  - Cannot compare in entirety
    - Must compare member by member



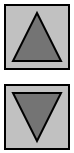
Outline

fig06_01.cpp
(1 of 3)

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

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

#include <iomanip>

using std::setfill;
using std::setw;

// structure definition
struct Time {
   int hour;     // 0-23 (24-hour clock format)
   int minute;   // 0-59
   int second;   // 0-59

}; // end struct Time

void printUniversal( const Time & );  // prototype
void printStandard( const Time & );   // prototype
```

Define structure type **`Time`** with three integer members.

Pass references to constant **`Time`** objects to eliminate copying overhead.



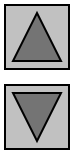
Outline

fig06_01.cpp
(2 of 3)

```
int main()
{
   Time dinnerTime;          //

   dinnerTime.hour = 18;    // set hour member of dinnerTime
   dinnerTime.minute = 30;  // set minute member of dinnerTime
   dinnerTime.second = 0;   // set second member of dinnerTime

   cout << "Dinner will be held at ";
   printUniversal( dinnerTime );
   cout << " universal time,\nwhich is ";
   printStandard( dinnerTime );
   cout << " standard time.\n";

   dinnerTime.hour = 29;   // set hour to invalid value
   dinnerTime.minute = 73;  // set minute to invalid value

   cout << "\nTime with invalid values: ";
   printUniversal( dinnerTime );
   cout << endl;

   return 0;

} // end main
```

Use dot operator to initialize structure members.

Direct access to data allows assignment of bad values.



Outline

fig06_01.cpp
(3 of 3)

```
// print time in universal-time format
void printUniversal( const Time &t )
{
   cout << setfill( '0' ) << setw( 2 ) << t.hour << ":"
        << setw( 2 ) << t.minute << ":"
        << setw( 2 ) << t.second;

} // end function printUniversal

// print time in standard-time format
void printStandard( const Time &t )
{
   cout << ( ( t.hour == 0 || t.hour == 12 ) ?
             12 : t.hour % 12 ) << ":" << setfill( '0' )
        << setw( 2 ) << t.minute << ":"
        << setw( 2 ) << t.second
        << ( t.hour < 12 ? " AM" : " PM" );

} // end function printStandard
```

Use parameterized stream manipulator **`setfill`**.

Use dot operator to access data members.

```
Dinner will be held at 18:30:00 universal time,
which is 6:30:00 PM standard time.

Time with invalid values: 29:73:00
```

6.5 Implementing a `Time` Abstract Data Type with a `class`

- Classes
  - Model objects
    - Attributes (data members)
    - Behaviors (member functions)
  - Defined using keyword **`class`**
  - Member functions
    - Methods
    - Invoked in response to messages
- Member access specifiers
  - **`public:`**
    - Accessible wherever object of class in scope
  - **`private:`**
    - Accessible only to member functions of class
  - **`protected:`**

6.5 Implementing a `Time` Abstract Data Type with a `class`

- Constructor function
  - Special member function
    - Initializes data members
    - Same name as class
  - Called when object instantiated
  - Several constructors
    - Function overloading
  - No return type

Outline

Class `Time` definition (1 of 1)

```
class Time {

public:
   Time();
   void setTime( int
   void printUniversal();
   void printStandard();

private:
   int hour;
   int minute;
   int second;

}; // end class Time
```

Definition with keywo

Class body star brace.

Function prototypes for **`public`** member functions.

Constructor has same name as class, **`Time`**, and no return type.

functions.

Definition terminates with semicolon.

6.5 Implementing a `Time` Abstract Data Type with a `class`

- Objects of class
  - After class definition
    - Class name new type specifier
      - C++ extensible language
    - Object, array, pointer and reference declarations
  - Example:

Class name becomes new type specifier.

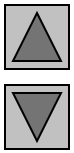
```
Time sunset;                    // object of type Time
Time arrayOfTimes[ 5 ];         // array of Time objects
Time *pointerToTime;            // pointer to a Time object
Time &dinnerTime = sunset;      // reference to a Time object
```

6.5 Implementing a `Time` Abstract Data Type with a `class`

- Member functions defined outside class
  - Binary scope resolution operator (`::`)
    - "Ties" member name to class name
    - Uniquely identify functions of particular class
    - Different classes can have member functions with same name
  - Format for defining member functions

```
ReturnType ClassName::MemberFunctionName( ){
   …
}
```

  - Does not change whether function **`public`** or **`private`**
- Member functions defined inside class
  - Do not need scope resolution operator, class name
  - Compiler attempts **`inline`**
    - Outside class, inline explicitly with keyword **`inline`**

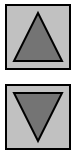


Outline

fig06_03.cpp
(1 of 5)

```
// Fig. 6.3: fig06_03.cpp
// Time class.
#include <iostream>

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

#include <iomanip>

using std::setfill;
using std::setw;

// Time abstract data type (ADT) definition
class Time {

public:
   Time();                          // constructor
   void setTime( int, int, int ); // set hour, minute, second
   void printUniversal();           // print universal-time format
   void printStandard();            // print standard-time format

```

Define class **`Time`**.



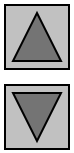
Outline

fig06_03.cpp
(2 of 5)

```
private:
   int hour;     // 0 - 23 (24-hour clock format)
   int minute;   // 0 - 59
   int second;   // 0 - 59

}; // end class Time

// Time constructor initializes each data me
// ensures all Time objects start in a consi
Time::Time()
{
   hour = minute = second = 0;

} // end Time constructor

// set new Time value using universal time, perform validity
// checks on the data values and set invalid values to zero
void Time::setTime( int h, int m, int s )
{
   hour = ( h >= 0 && h < 24 ) ? h : 0;
   minute = ( m >= 0 && m < 60 ) ? m : 0;
   second = ( s >= 0 && s < 60 ) ? s : 0;

} // end function setTime

```

Constructor initializes **`private`** data members to **`0`**.

`public` member function checks parameter values for validity before setting **`private`** data members.



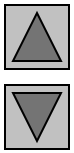
Outline

fig06_03.cpp
(3 of 5)

```
// print Time in universal format
void Time::printUniversal()
{
   cout << setfill( '0' ) << setw( 2 ) << hour << ":"
        << setw( 2 ) << minute << ":"
        << setw( 2 ) << second;

} // end function printUniversal

// print Time in standard format
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 print

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

```

No arguments (implicitly "know" purpose is to print data members); member function calls more concise.

Declare variable **t** to be object of class **Time**.



Outline

fig06_03.cpp
(4 of 5)

```
   // 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 val
   cout << "\n\nUniversal time after
   t.printUniversal();   // 13:27:06

   cout << "\nStandard time after se
   t.printStandard();    // 1:27:06

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

   // output t's values after specifying invalid values
   cout << "\n\nAfter attempting invalid settings:"
        << "\nUniversal time: ";
   t.printUniversal();   // 00:00:00

```

Invoke **`public`** member functions to print time.

Set data members using **`public`** member function.

Attempt to set data members to invalid values using **`public`** member function.

```
   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
```

Data members set to **0** after attempting invalid settings.

fig06_03.cpp (5 of 5)

fig06_03.cpp output (1 of 1)

6.5 Implementing a `Time` Abstract Data Type with a `class`

- Destructors
  - Same name as class
    - Preceded with tilde (~)
  - No arguments
  - Cannot be overloaded
  - Performs “termination housekeeping”

6.5 Implementing a `Time` Abstract Data Type with a `class`

- Advantages of using classes
  - Simplify programming
  - Interfaces
    - Hide implementation
  - Software reuse
    - Composition (aggregation)
      - Class objects included as members of other classes
    - Inheritance
      - New classes derived from old

6.6 Class Scope and Accessing Class Members

- Class scope
  - Data members, member functions
  - Within class scope
    - Class members
      - Immediately accessible by all member functions
      - Referenced by name
  - Outside class scope
    - Referenced through handles
      - Object name, reference to object, pointer to object
- File scope
  - Nonmember functions

6.6 Class Scope and Accessing Class Members

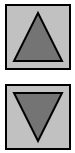
- Function scope
  - Variables declared in member function
  - Only known to function
  - Variables with same name as class-scope variables
    - Class-scope variable “hidden”
      - Access with scope resolution operator (`::`)

```
ClassName::classVariableName
```

  - Variables only known to function they are defined in
  - Variables are destroyed after function completion

6.6 Class Scope and Accessing Class Members

- Operators to access class members
  - Identical to those for **`struct`**s
  - Dot member selection operator (**`.`**)
    - Object
    - Reference to object
  - Arrow member selection operator (**`->`**)
    - Pointers

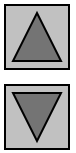


Outline

fig06_04.cpp
(1 of 2)

```
// Fig. 6.4: fig06_04.cpp
// Demonstrating the class member access operators . and ->
//
// CAUTION: IN FUTURE EXAMPLES WE AVOID PUBLIC DATA!
#include <iostream>

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

// class Count definition
class Count {

public:
   int x;

   void print()
   {
      cout << x << endl;
   }

}; // end class Count

```

Data member **`x public`** to illustrate class member access operators; typically data members **`private`**.



Outline

fig06_04.cpp (2 of 2)

fig06_04.cpp output (1 of 1)

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

   cout << "Assign 1 to x and prin
   counter.x = 1;      // assign 1 t
   counter.print();    // call membe

   cout << "Assign 2 to x and print u
   counterRef.x = 2;   // assign 2 to
   counterRef.print(); // call member

   cout << "Assign 3 to x and print us
   counterPtr->x = 3;  // assign 3 to data member x
   counterPtr->print(); // call member function print

   return 0;

} // end main
```

Use dot member selection operator for **`counter`** object.

Use dot member selection operator for **`counterRef`** reference to object.

Use arrow member selection operator for **`counterPtr`** pointer to object.

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

6.7 Separating Interface from Implementation

- Separating interface from implementation
  - Advantage
    - Easier to modify programs
  - Disadvantage
    - Header files
      - Portions of implementation
        - Inline member functions
      - Hints about other implementation
        - private members
    - Can hide more with proxy class

6.7 Separating Interface from Implementation

- Header files
  - Class definitions and function prototypes
  - Included in each file using class
    - `#include`
  - File extension `.h`
- Source-code files
  - Member function definitions
  - Same base name
    - Convention
  - Compiled and linked

Outline

time1.h (1 of 1)

```
// Fig. 6.5: time1.h
// Declaration of class Time.
// Member functions are defined in t

// prevent multiple inclusions of header file
#ifndef TIME1_H
#define TIME1_H

// Time abstract d
class Time {

public:
   Time();                          // constructor
   void setTime( int, int, int ); // set hour, minute, second
   void printUniversal();         // print universal-time format
   void printStandard();          // print standard-time format

private:
   int hour;     // 0 - 23 (24-hour clock format)
   int minute;   // 0 - 59
   int second;   // 0 - 59

}; // end class Time

#endif
```

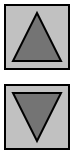
Preprocessor code to prevent multiple inclusions.

Code between these directives

"If not d

fines

Naming convention: header file name with underscore replacing period.



Outline

time1.cpp (1 of 3)

```
// Fig. 6.6: time1.cpp
// Member-function definitions for class Time.
#include <iostream>

using std::cout;

#include <iomanip>

using std::setfill;
using std::setw;

// include definition of class Time from time1.h
#include "time1.h"

// Time constructor initializes each data member to zero.
// Ensures all Time objects
Time::Time()
{
   hour = minute = second =

} // end Time constructor

```

Include header file **`time1.h`**.

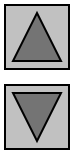
Name of header file enclosed in quotes; angle brackets cause preprocessor to assume header part of C++ Standard Library.

time1.cpp (2 of 3)

```
// Set new Time value using universal time. Perform validity
// checks on the data values. Set invalid values to zero.
void Time::setTime( int h, int m, int s )
{
   hour = ( h >= 0 && h < 24 ) ? h : 0;
   minute = ( m >= 0 && m < 60 ) ? m : 0;
   second = ( s >= 0 && s < 60 ) ? s : 0;

} // end function setTime

// print Time in universal format
void Time::printUniversal()
{
   cout << setfill( '0' ) << setw( 2 ) << hour << ":"
        << setw( 2 ) << minute << ":"
        << setw( 2 ) << second;

} // end function printUniversal

```

time1.cpp (3 of 3)

```
// print Time in standard format
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
```



Outline

fig06_07.cpp
(1 of 2)

```
// Fig. 6.7: fig06_07.cpp
// Program to test class Time.
// NOTE: This file must be compiled with time1.cpp.
#include <iostream>

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

// include definition of class Time
#include "time1.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

```

Include header file **`time1.h`** to ensure correct creation/manipulation and determine size of **`Time`** class object.

```
   // 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

   // 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
```

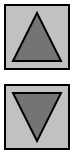
```
The initial universal time is 00:00:00
The initial standard time is 12:00:00 AM

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

fig06_07.cpp (2 of 2)

fig06_07.cpp output (1 of 1)

6.8 Controlling Access to Members

- Access modes
  - **`private`**
    - Default access mode
    - Accessible to member functions and **`friend`**s
  - **`public`**
    - Accessible to any function in program with handle to class object
  - **`protected`**
    - Chapter 9



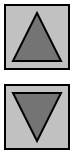
Outline

fig06_08.cpp
(1 of 1)

```
// Fig. 6.8: fig06_08.cpp
// Demonstrate errors resulting from attempts
// to access private class members.
#include <iostream>

using std::cout;

// include definition of class Time from time1.h
#include "time1.h"

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


   t.hour = 7;  // error: 'Time

   // error: 'Time::minute' is not accessible
   cout << "minute = " << t.minute;

   return 0;

} // end main
```

Recall data member **`hour`** is **`private`**; attempts to access **`private`** members results in error.

Data member **`minute`** also **`private`**; attempts to access **`private`** members produces error.



Outline

fig06_08.cpp output (1 of 1)

```
D:\cpphtp4_examples\ch06\Fig6_06\Fig06_06.cpp(16) : error C2248:
   'hour' : cannot access private member declared in class 'Time'
D:\cpphtp4_examples\ch06\Fig6_06\Fig06_06.cpp(19) : error C2248:
   'minute' : cannot access private member declared in class 'Time'
```

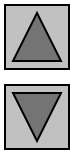
Errors produced by attempting to access **`private`** members.

6.8 Controlling Access to Members

- Class member access
  - Default **`private`**
  - Explicitly set to **`private`**, **`public`**, **`protected`**
- **`struct`** member access
  - Default **`public`**
  - Explicitly set to **`private`**, **`public`**, **`protected`**
- Access to class's **`private`** data
  - Controlled with access functions (accessor methods)
    - Get function
      - Read **`private`** data
    - Set function
      - Modify **`private`** data

6.9 Access Functions and Utility Functions

- Access functions
  - **`public`**
  - Read/display data
  - Predicate functions
    - Check conditions
- Utility functions (helper functions)
  - **`private`**
  - Support operation of **`public`** member functions
  - Not intended for direct client use



Outline

salesp.h (1 of 1)

```
// Fig. 6.9: salesp.h
// SalesPerson class definition.
// Member functions defined in salesp.cpp.
#ifndef SALESP_H
#define SALESP_H

class SalesPerson {

public:
   SalesPerson();                  // construct
   void getSalesFromUser();        // input sales from keyboard
   void setSales( int, double );   // set sales
   void printAnnualSales();        // summarize

private:
   double totalAnnualSales();      // utility function
   double sales[ 12 ];             // 12 monthly sales figures

}; // end class SalesPerson

#endif
```

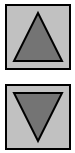
Set access function performs validity checks.

private utility function.

Outline

salesp.cpp (1 of 3)

```
// Fig. 6.10: salesp.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 class definition from salesp.h
#include "salesp.h"

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

} // end SalesPerson constructor

```



Outline

salesp.cpp (2 of 3)

```
// 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 subscri
void SalesPerson::setSales( int month, dou
{
   // 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;
```

Set access function performs validity checks.

Outline

salesp.cpp (3 of 3)

```
} // end function setSales

// print total annual sales (with 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 ];

   return total;

} // end function totalAnnualSales
```

`private` utility function to help function **`printAnnualSales`**; encapsulates logic of manipulating **`sales`** array.

Outline

fig06_11.cpp
(1 of 1)

```
// Fig. 6.11: fig06_11.cpp
// Demonstrating a utility function.
// Compile this program with salesp.cpp

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

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

   s.getSalesFromUser();  // note simple sequential co
   s.printAnnualSales();  // control structures in main

   return 0;

} // end main
```

Simple sequence of member function calls; logic encapsulated in member functions.

Outline

fig06_11.cpp output (1 of 1)

```
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
```

6.10 Initializing Class Objects: Constructors

- Constructors
  - Initialize data members
    - Or can set later
  - Same name as class
  - No return type
- Initializers
  - Passed as arguments to constructor
  - In parentheses to right of class name before semicolon

```
Class-type ObjectName( value1,value2,…);
```

6.11 Using Default Arguments with Constructors

- Constructors
  - Can specify default arguments
  - Default constructors
    - Defaults all arguments

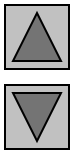
      OR
    - Explicitly requires no arguments
    - Can be invoked with no arguments
    - Only one per class

Outline

time2.h (1 of 1)

```
// Fig. 6.12: time2.h
// Declaration of class Time.
// Member functions defined in time2.cpp.

// prevent multiple inclusions of header file
#ifndef TIME2_H
#define TIME2_H

// Time abstract data type definition
class Time {

public:
   Time( int = 0, int = 0, int = 0); // default constructor
   void setTime( int, int, int ); // set hour, minute, second
   void printUniversal();         // print universal-time format
   void printStandard();          // print standard-time format

private:
   int hour;     // 0 - 23 (24-hour clock format)
   int minute;   // 0 - 59
   int second;   // 0 - 59

}; // end class Time

#endif
```

Default constructor specifying all arguments.



Outline

time2.cpp (1 of 3)

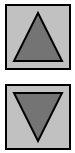
```
// Fig. 6.13: time2.cpp
// Member-function definitions for class Time.
#include <iostream>

using std::cout;

#include <iomanip>

using std::setfill;
using std::setw;

// include definition of class Time from time2.h
#include "time2.h"

// Time constructor initializes each data member to zero;
// ensures all 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

```

Constructor calls **`setTime`** to validate passed (or default) values.

Outline

time2.cpp (2 of 3)

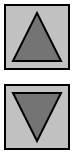
```
// set new Time value using universal time, perform validity
// checks on the data values and set invalid values to zero
void Time::setTime( int h, int m, int s )
{
   hour = ( h >= 0 && h < 24 ) ? h : 0;
   minute = ( m >= 0 && m < 60 ) ? m : 0;
   second = ( s >= 0 && s < 60 ) ? s : 0;

} // end function setTime

// print Time in universal format
void Time::printUniversal()
{
   cout << setfill( '0' ) << setw( 2 ) << hour << ":"
        << setw( 2 ) << minute << ":"
        << setw( 2 ) << second;

} // end function printUniversal

```



Outline

time2.cpp (3 of 3)

```
// print Time in standard format
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
```



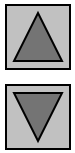
Outline

fig06_14.cpp
(1 of 2)

```cpp
// Fig. 6.14: fig06_14.cpp
// Demonstrating a default constructor for class Time.
#include <iostream>

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

// include definition of class Time from time2.h
#include "time2.h"

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

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

```

Initialize **Time** objects using default arguments.

Initialize **Time** object with invalid values; validity checking will set values to **0**.



Outline

fig06_14.cpp
(2 of 2)

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

   cout << "\n\nhour and minute specified; default second:\n  ";
   t3.printUniversal();  // 21:34:00
   cout << "\n  ";
   t3.printStandard();   // 9:34:00 PM

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

   cout << "\n\nall invalid values specified:\n  ";
   t5.printUniversal();  // 00:00:00
   cout << "\n  ";
   t5.printStandard();   // 12:00:00 AM
   cout << endl;

   return 0;

} // end main
```

t5 constructed with invalid arguments; values set to **0**.

Outline

fig06_14.cpp output (1 of 1)

```
Constructed with:

all default arguments:
  00:00:00
  12:00:00 AM

hour specified; default minute and second:
  02:00:00
  2:00:00 AM

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

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

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

6.12 Destructors

- Destructors
  - Special member function
  - Same name as class
    - Preceded with tilde (~)
  - No arguments
  - No return value
  - Cannot be overloaded
  - Performs “termination housekeeping”
    - Before system reclaims object’s memory
      - Reuse memory for new objects
  - No explicit destructor
    - Compiler creates “empty” destructor”

6.13 When Constructors and Destructors Are Called

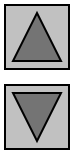
- Constructors and destructors
  - Called implicitly by compiler
- Order of function calls
  - Depends on order of execution
    - When execution enters and exits scope of objects
  - Generally, destructor calls reverse order of constructor calls

6.13 When Constructors and Destructors Are Called

- Order of constructor, destructor function calls
  - Global scope objects
    - Constructors
      - Before any other function (including **`main`**)
    - Destructors
      - When **`main`** terminates (or **`exit`** function called)
      - Not called if program terminates with **`abort`**
  - Automatic local objects
    - Constructors
      - When objects defined
        - Each time execution enters scope
    - Destructors
      - When objects leave scope
        - Execution exits block in which object defined
      - Not called if program ends with **`exit`** or **`abort`**

6.13 When Constructors and Destructors Are Called

- Order of constructor, destructor function calls
  - **`static`** local objects
    - Constructors
      - Exactly once
      - When execution reaches point where object defined
    - Destructors
      - When **`main`** terminates or **`exit`** function called
      - Not called if program ends with **`abort`**



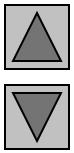
Outline

create.h (1 of 1)

```
// Fig. 6.15: create.h
// Definition of class CreateAndDestroy.
// Member functions defined in create.cpp.
#ifndef CREATE_H
#define CREATE_H

class CreateAndDestroy {

public:
   CreateAndDestroy( int, char * );  // constructor
   ~CreateAndDestroy();

private:
   int objectID;
   char *message;

}; // end class CreateAndDestroy

#endif
```

Constructor and destructor member functions.

`private` members to show order of constructor, destructor function calls.

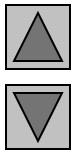


Outline

create.cpp (1 of 2)

```
// Fig. 6.16: create.cpp
// Member-function definitions for class CreateAndDestroy
#include <iostream>

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

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

// constructor
CreateAndDestroy::CreateAndDestroy(
   int objectNumber, char *messagePtr )
{
   objectID = objectNumber;
   message = messagePtr;

   cout << "Object " << objectID << "   constructor runs   "
        << message << endl;

} // end CreateAndDestroy constructor

```

Output message to demonstrate timing of constructor function calls.

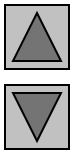


Outline

create.cpp (2 of 2)

```
// destructor
CreateAndDestroy::~CreateAndDestroy()
{
   // the following line is for pedag
   cout << ( objectID == 1 || objectI

   cout << "Object " << objectID << "   destructor runs    "
        << message << endl;

} // end ~CreateAndDestroy destructor
```

Output message to demonstrate timing of destructor function calls.



Outline

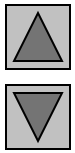
fig06_17.cpp
(1 of 3)

```
// Fig. 6.17: fig06_17.cpp
// Demonstrating the order in which constructors and
// destructors are called.
#include <iostream>

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

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

void create( void );   // prototype

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

int main()
{
   cout << "\nMAIN FUNCTION: EXECUTION

   CreateAndDestroy second( 2, "(local automatic in main)" );

   static CreateAndDestroy third(
      3, "(local static in main)" );

```

Create variable with global scope.

Create local automatic object.

Create **static** local object.



Outline

fig06_17.cpp
(2 of 3)

```
   create();  // call function to create objects

   cout << "\nMAIN FUNCTION: EXECUTION RESUMES" << endl;

   CreateAndDestroy fourth(                                  " );

   cout << "\nMAIN FUNCTION: EXECUTION ENDS" << endl;

   return 0;

} // end main

// function to create objects
void create( void )
{
   cout << "\nCREATE FUNCTIO

   CreateAndDestroy fifth( 5                                 )" );

   static CreateAndDestroy s
      6, "(local static in c

   CreateAndDestroy seventh(
      7, "(local automatic in create)" );

```

Create local automatic objects.

Create local automatic object.

Create local automatic object in function.

Create **static** local object in function.

Create local automatic object in function.

```
   cout << "\nCREATE FUNCTION: EXECUTION ENDS\" << endl;

} // end function create
```


fig06_17.cpp
(3 of 3)

Outline

fig06_17.cpp output (1 of 1)

```
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)
```

Local **static** object exists

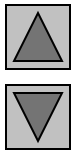
Global object constructed

Local automatic objects

Local **static** object constructed on first function call and destroyed after **main** execution ends.

6.14 Using *Set* and *Get* Functions

- Set functions
  - Perform validity checks before modifying **`private`** data
  - Notify if invalid values
  - Indicate with return values
- Get functions
  - “Query” functions
  - Control format of data returned



Outline

time3.h (1 of 2)

```
// Fig. 6.18: time3.h
// Declaration of class Time.
// Member functions defined in time3.cpp

// prevent multiple inclusions of header file
#ifndef TIME3_H
#define TIME3_H

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
   void setMinute( int ); // set minute
   void setSecond( int ); // set second

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

```

Set functions.

Get functions.

Outline

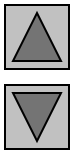
time3.h (2 of 2)

```
   void printUniversal(); // output universal-time format
   void printStandard();  // output standard-time format

private:
   int hour;              // 0 - 23 (24-hour clock format)
   int minute;            // 0 - 59
   int second;            // 0 - 59

}; // end clas Time

#endif
```

Outline

time3.cpp (1 of 4)

```cpp
// Fig. 6.19: time3.cpp
// Member-function definitions for Time class.
#include <iostream>

using std::cout;

#include <iomanip>

using std::setfill;
using std::setw;

// include definition of class Time from time3.h
#include "time3.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 );

} // end Time constructor

```



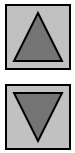
Outline

time3.cpp (2 of 4)

```
// set hour, minute and second values
void Time::setTime( int h, int m, int s )
{
   setHour( h );
   setMinute( m );
   setSecond( s );

} // end function setTime

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

} // end function setHour

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

} // end function setMinute

```

Call set functions to perform validity checking.

Set functions perform validity checks before modifying data.



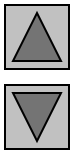
Outline

time3.cpp (3 of 4)

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

} // 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

```

Set function performs validity checks before modifying data.

Get functions allow client to read data.

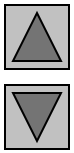


Outline

time3.cpp (4 of 4)

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

} // end function getSecond

// print Time in universal format
void Time::printUniversal()
{
   cout << setfill( '0' ) << setw( 2 ) << hour << ":"
        << setw( 2 ) << minute << ":"
        << setw( 2 ) << second;

} // end function printUniversal

// print Time in standard format
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
```

Get function allows client to read data.



Outline

fig06_20.cpp
(1 of 3)

```
// Fig. 6.20: fig06_20.cpp
// Demonstrating the Time class set and get functions
#include <iostream>

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

// include definition of class Time from time3.h
#include "time3.h"

void incrementMinutes( Time &, const int );  // prototype

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

   // set time using individual set functions
   t.setHour( 17 );     // set hour to valid value
   t.setMinute( 34 );   // set minute to valid value
   t.setSecond( 25 );   // set second to valid value

```

Invoke set functions to set valid values.

Outline

```
   // use get functions to obtain hour, minute and second
   cout << "Result of setting all valid values:\n"
        << "  Hour: " << t.getHour()
        << "  Minute: " << t.getMinute()
        << "  Second: " << t.getSecond();

   // set time using individual set functions
   t.setHour( 234 );    // invalid hour set to 0
   t.setMinute( 43 );   // set minute to valid value
   t.setSecond( 6373 ); // invalid second set to 0

   // display hour, minute and second after setting
   // invalid hour and second values
   cout << "\n\nResult of attempting to set invalid hour and"
        << " second:\n  Hour: " << t.getHour()
        << "  Minute: " << t.getMinute()
        << "  Second: " << t.getSecond() << "\n\n";

   t.setTime( 11, 58, 0 );    // set time
   incrementMinutes( t, 3 );  // increment t's minute by 3

   return 0;

} // end main
```

Attempt to set invalid values using set functions.

Invalid values result in setting data members to **0**.

Modify data members using function **setTime**.

Outline

fig06_20.cpp

```
// add specified number of minutes to a Time object
void incrementMinutes( Time &tt, const int count )
{
   cout << "Incrementing minute " << count
        << " times:\nStart time: ";
   tt.printStandard();

   for ( int i = 0; i < count; i++ ) {
      tt.setMinute( ( tt.getMinute() + 1 ) % 60 );

      if ( tt.getMinute() == 0 )
         tt.setHour( ( tt.getHour() + 1 ) % 24);

      cout << "\nminute + 1: ";
      tt.printStandard();

   } // end for

   cout << endl;

} // end function incrementMinutes
```

Using get functions to read data and set functions to modify data.

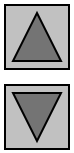
fig06_20.cpp output (1 of 1)

```
Result of setting all valid values:
  Hour: 17  Minute: 34  Second: 25

Result of attempting to set invalid hour and second:
  Hour: 0  Minute: 43  Second: 0

Incrementing minute 3 times:
Start time: 11:58:00 AM
minute + 1: 11:59:00 AM
minute + 1: 12:00:00 PM
minute + 1: 12:01:00 PM
```

Attempting to set data members with invalid values results in error message and members set to **0**.

6.15 Subtle Trap: Returning a Reference to a `private` Data Member

- Reference to object
  - Alias for name of object
  - Lvalue
    - Can receive value in assignment statement
      - Changes original object
- Returning references
  - **`public`** member functions can return non-**`const`** references to **`private`** data members
    - Client able to modify **`private`** data members



Outline

time4.h (1 of 1)

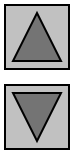
```
// Fig. 6.21: time4.h
// Declaration of class Time.
// Member functions defined in time4.cpp

// prevent multiple inclusions of header file
#ifndef TIME4_H
#define TIME4_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
```

Function to demonstrate effects of returning reference to **`private`** data member.

Outline

time4.cpp (1 of 2)

```
// Fig. 6.22: time4.cpp
// Member-function definitions for Time class.

// include definition of class Time from time4.h
#include "time4.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 );

} // 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;
   minute = ( m >= 0 && m < 60 ) ? m : 0;
   second = ( s >= 0 && s < 60 ) ? s : 0;

} // end function setTime

```

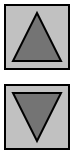


Outline

time4.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 )

   return hour;  // DANGEROUS reference return

} // end function badSetHour
```

Return reference to **`private`** data member **`hour`**.



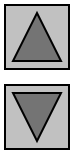
Outline

fig06_23.cpp
(1 of 2)

```
// Fig. 6.23: fig06_23.cpp
// Demonstrating a public member function that
// returns a reference to a private data member.
#include <iostream>

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

// include definition of class Time from time4.h
#include "time4.h"

int main()
{
   Time t;

   // store in hourRef the reference returned by badSe
   int &hourRef = t.badSetHour( 20 );

   cout << "Hour before modificati

   // use hourRef to set invalid v
   hourRef = 30;

   cout << "\nHour after modification: " << t.getHour();

```

`badSetHour` returns reference to **`private`** data member **`hour`**.

Reference allows setting of **`private`** data member **`hour`**.



Outline

fig06_23.cpp (2 of 2)

fig06_23.cpp output (1 of 1)

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

   cout << "\n\n*********************************\n"
        << "POOR PROGRAMMING PRACTIC
        << "badSetHour as an lvalue,
        << t.getHour()
        << "\n*********************************" << endl;

   return 0;

} // end main
```

Can use function call as lvalue to set invalid value.

```
Hour before modification: 20
Hour after modification: 30

*********************************
POOR PROGRAMMING PRACTICE!!!!!!!
badSetHour as an lvalue, Hour: 74
*********************************
```

Returning reference allowed invalid setting of **`private`** data member **`hour`**.

6.16 Default Memberwise Assignment

- Assigning objects
  - Assignment operator (**=**)
    - Can assign one object to another of same type
    - Default: memberwise assignment
      - Each right member assigned individually to left member
- Passing, returning objects
  - Objects passed as function arguments
  - Objects returned from functions
  - Default: pass-by-value
    - Copy of object passed, returned
      - Copy constructor
        - Copy original values into new object

Outline

fig06_24.cpp
(1 of 3)

```
// Fig. 6.24: fig06_24.cpp
// Demonstrating that class objects can be assigned
// to each other using default memberwise assignment.
#include <iostream>

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

// class Date definition
class Date {

public:
   Date( int = 1, int = 1, int = 1990 ); // default constructor
   void print();

private:
   int month;
   int day;
   int year;

}; // end class Date

```

Outline

fig06_24.cpp
(2 of 3)

```
// Date constructor with no range checking
Date::Date( int m, int d, int y )
{
   month = m;
   day = d;
   year = y;

} // end Date constructor

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

} // end function print

int main()
{
   Date date1( 7, 4, 2002 );
   Date date2;  // date2 defaults to 1/1/1990

```

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

   date2 = date1;   // default mem

   cout << "\n\nAfter default memberwise assignment, date2 = ";
   date2.print();
   cout << endl;

   return 0;

} // end main
```

Default memberwise assignment assigns each member of **`date1`** individually to each member of **`date2`**.

```
date1 = 7-4-2002
date2 = 1-1-1990

After default memberwise assignment, date2 = 7-4-2002
```

Outline

fig06_24.cpp (3 of 3)

fig06_24.cpp output (1 of 1)

6.17 Software Reusability

- Software reusability
  - Class libraries
    - Well-defined
    - Carefully tested
    - Well-documented
    - Portable
    - Widely available
  - Speeds development of powerful, high-quality software
    - Rapid applications development (RAD)
  - Resulting problems
    - Cataloging schemes
    - Licensing schemes
    - Protection mechanisms