

計算機程式 OBJECT-ORIENTED PROGRAMMING 物件導向程式設計 DME1584

Lecture #05

Classes

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**

Structure Definitions

3

- Structures

- Aggregate data types built using elements of other types

```

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

- Structure member naming

- In same **struct**: must have unique names
- In different **structs**: can share name

- **struct** definition must end with semicolon

Structure Definitions

4

- 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 Time {
    int hour;
    int minute;
    int second;
    struct Time *Next_Time;
};
  
```

Structure Definitions

5

- **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;`

Accessing Structure Members

6

- 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 .

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

```

1  // Fig. 6.1: fig06_01.cpp
2  // Create a structure, set its members, and print it.
3  #include <iostream>
4
5  using std::cout;
6  using std::endl;
7
8  #include <iomanip>
9
10 using std::setfill;
11 using std::setw;
12
13 // structure definition
14 struct Time {
15     int hour;    // 0-23 (24-hour clock format)
16     int minute; // 0-59
17     int second; // 0-59
18 }; // end struct Time
19
20
21 void printUniversal( const Time & ); // prototype
22 void printStandard( const Time & ); // prototype
23

```

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

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

Outline

fig06_01.cpp
(1 of 3)

9

Outline

fig06_01.cpp
(2 of 3)

24 int main()
25 {
26 Time dinnerTime; // variable of new type Time
27
28 dinnerTime.hour = 18; // set hour member of dinnerTime
29 dinnerTime.minute = 30; // set minute member of dinnerTime
30 dinnerTime.second = 0; // set second member of dinnerTime
31
32 cout << "Dinner will be held at ";
33 printUniversal(dinnerTime);
34 cout << " universal time,\nwhich is ";
35 printStandard(dinnerTime);
36 cout << " standard time.\n";
37
38 dinnerTime.hour = 29; // set hour to invalid value
39 dinnerTime.minute = 73; // set minute to invalid value
40
41 cout << "\nTime with invalid values: ";
42 printUniversal(dinnerTime);
43 cout << endl;
44
45 return 0;
46
47 } // end main
48

Use dot operator to initialize structure members.

Direct access to data allows assignment of bad values.

10

Outline

fig06_01.cpp
(3 of 3)

49 // print time in universal-time format
50 void printUniversal(const Time &t)
51 {
52 cout << setfill('0') << setw(2) << t.hour << ":"
53 << setw(2) << t.minute << ":"
54 << setw(2) << t.second;
55 } // end function printUniversal
56
57 // print time in standard-time format
58 void printStandard(const Time &t)
59 {
60 cout << ((t.hour == 0 || t.hour == 12) ?
61 12 : t.hour % 12) << ":" << setfill('0')
62 << setw(2) << t.minute << ":"
63 << setw(2) << t.second
64 << (t.hour < 12 ? " AM" : " PM");
65
66 } // 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

5

Implementing a Time Abstract Data Type with a class

11

- 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:**

Implementing a Time Abstract Data Type with a class

12

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

13
 Outline
 Class Time definition
 (1 of 1)

```

1  class Time {
2
3  public:
4      Time();
5      void setTime( int, int, int );
6      void printUniversal();
7      void printStandard();
8
9  private:
10     int hour;
11     int minute;
12     int second;
13
14 } // end class Time
  
```

Definition with keyword **public**.

Class body starts with brace.

Function prototypes for **public** member functions.

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

Definition terminates with semicolon.

functions.

Implementing a Time Abstract Data Type with a class

14

- 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
          
```

Implementing a Time Abstract Data Type with a class

15

- 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**

```

1  // Fig. 6.3: fig06_03.cpp
2  // Time class.
3  #include <iostream>
4
5  using std::cout;
6  using std::endl;
7
8  #include <iomanip>
9
10 using std::setfill;
11 using std::setw;
12
13 // Time abstract data type (ADT) definition
14 class Time {
15
16 public:
17     Time(); // constructor
18     void setTime( int, int, int ); // set hour, minute, second
19     void printUniversal(); // print universal-time format
20     void printStandard(); // print standard-time format
21

```

Define class **Time**.



Outline

fig06_03.cpp
(1 of 5)

16

17
[Outline](#)
 fig06_03.cpp
 (2 of 5)

```

22 private:
23     int hour;      // 0 - 23 (24-hour clock format)
24     int minute;    // 0 - 59
25     int second;    // 0 - 59
26
27 }; // end class Time
28
29 // Time constructor initializes each data member to 0
30 // ensures all Time objects start in a consistent state
31 Time::Time()
32 {
33     hour = minute = second = 0;
34 } // end Time constructor
35
36 // set new Time value using universal time, perform validity
37 // checks on the data values and set invalid values to zero
38 void Time::setTime( int h, int m, int s )
39 {
40     hour = ( h >= 0 && h < 24 ) ? h : 0;
41     minute = ( m >= 0 && m < 60 ) ? m : 0;
42     second = ( s >= 0 && s < 60 ) ? s : 0;
43 } // end function setTime
44
45
46
  
```

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

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

18
[Outline](#)
 fig06_03.cpp
 (3 of 5)

```

47 // print Time in universal format
48 void Time::printUniversal()
49 {
50     cout << setfill( '0' ) << setw( 2 ) << hour << ":"
51         << setw( 2 ) << minute << ":"
52         << setw( 2 ) << second;
53 } // end function printUniversal
54
55 // print Time in standard format
56 void Time::printStandard()
57 {
58     cout << ( ( hour == 0 || hour == 12 ) ? 12 : hour % 12 )
59         << ":" << setfill( '0' ) << setw( 2 ) << minute
60         << ":" << setw( 2 ) << second
61         << ( hour < 12 ? " AM" : " PM" );
62 } // end function printStandard
63
64 // declare variable t to be object of class Time
65
66 int main()
67 {
68     Time t; // instantiate object t of class Time
69 }
  
```

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

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

The image shows a C++ program snippet with line numbers 70 to 92. The code uses a `Time` object `t` to demonstrate various methods. Annotations with arrows point to specific lines:

- Line 74: `cout << "\nThe initial standard time is ";` is annotated with "Invoke **public** member functions to print time."
- Line 77: `t.setTime(13, 27, 6);` is annotated with "Set data members using **public** member function."
- Line 84: `t.setTime(99, 99, 99);` is annotated with "Attempt to set data members to invalid values using **public** member function."

The program output, shown in blue text, is:

```
The initial universal time is ";
00:00:00
The initial standard time is
12:00:00 AM
13, 27, 6
13:27:06
1:27:06
After attempting invalid settings:
00:00:00
```

Annotations:

- 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.

Navigation icons (back, forward, search, etc.) are visible in the top right corner.

The screenshot displays a C++ program and its execution output. The program defines a `Time` class with static data members `UniversalTime` and `StandardTime`, and methods `printStandard`, `setTime`, and `main`. The output shows the initial times, the effect of `setTime`, and that invalid settings are ignored.

```
93     cout << "\nStandard time: ";
94     t.printStandard();           // 12:00:00 AM
95     cout << endl;
96
97     return 0;
98
99 } // 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.

Navigation icons: back, forward, search, etc.

Outline

fig06_03.cpp (5 of 5)

fig06_03.cpp output (1 of 1)

Implementing a Time Abstract Data Type with a class

21

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

Implementing a Time Abstract Data Type with a class

22

- 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

23

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

24

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

Class Scope and Accessing Class Members

25

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

```

1 // Fig. 6.4: fig06_04.cpp
2 // Demonstrating the class member access operators . and ->
3 //
4 // CAUTION: IN FUTURE EXAMPLES WE AVOID PUBLIC DATA!
5 #include <iostream>
6
7 using std::cout;
8 using std::endl;
9
10 // class Count definition
11 class Count {
12
13 public:
14     int x;
15
16     void print()
17     {
18         cout << x << endl;
19     }
20
21 }; // end class Count
22

```

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



Outline

26

fig06_04.cpp
(1 of 2)

```

23 int main()
24 {
25     Count counter;           // create counter object
26     Count *counterPtr = &counter; // create pointer to counter
27     Count &counterRef = counter; // create reference to counter
28
29     cout << "Assign 1 to x and print using the object's name: ";
30     counter.x = 1;           // assign 1 to data member x
31     counter.print();         // call member function print
32
33     cout << "Assign 2 to x and print using a reference: ";
34     counterRef.x = 2;        // assign 2 to data member x
35     counterRef.print();      // call member function print
36
37     cout << "Assign 3 to x and print using a pointer: ";
38     counterPtr->x = 3;        // assign 3 to data member x
39     counterPtr->print();      // call member function print
40
41     return 0;
42 } // end main

```

Outline

fig06_04.cpp
(2 of 2)

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

Separating Interface from Implementation

28

- 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

29

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

```

1 // Fig. 6.5: time1.h
2 // Declaration of class Time.
3 // Member functions are defined in time1.cpp
4
5 // prevent multiple inclusions of header file
6 #ifndef TIME1_H
7 #define TIME1_H
8
9 // Time abstract data type definition
10 class Time {
11
12 public:
13     Time(); // constructor
14     void setTime( int, int, int ); // set hour, minute, second
15     void printUniversal(); // print universal-time format
16     void printStandard(); // print standard-time format
17
18 private:
19     int hour; // 0 - 23 (24-hour clock format)
20     int minute; // 0 - 59
21     int second; // 0 - 59
22
23 }; // end class Time
24
25 #endif

```

“If not defined”

Preprocessor directive defines name **TIME1_H**

Preprocessor code to prevent multiple inclusions.

Naming convention: header file name with underscore replacing period.

Code between these directives not included if name **TIME1_H** already defined.

Outline

30

```

1 // Fig. 6.6: time1.cpp
2 // Member-function definitions for class Time.
3 #include <iostream>
4
5 using std::cout;
6
7 #include <iomanip>
8
9 using std::setfill;
10 using std::setw;
11
12 // include definition of class Time from time1.h
13 #include "time1.h"
14
15 // Time constructor initializes each data member to zero.
16 // Ensures all Time objects start in a consistent state.
17 Time::Time()
18 {
19     hour = minute = second = 0;
20 }
21 // end Time constructor
22

```

[Outline](#)

time1.cpp (1 of 3)

31

Include header file
time1.h.

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

```

23 // Set new Time value using universal time. Perform validity
24 // checks on the data values. Set invalid values to zero.
25 void Time::setTime( int h, int m, int s )
26 {
27     hour = ( h >= 0 && h < 24 ) ? h : 0;
28     minute = ( m >= 0 && m < 60 ) ? m : 0;
29     second = ( s >= 0 && s < 60 ) ? s : 0;
30
31 } // end function setTime
32
33 // print Time in universal format
34 void Time::printUniversal()
35 {
36     cout << setfill( '0' ) << setw( 2 ) << hour << ":"
37         << setw( 2 ) << minute << ":"
38         << setw( 2 ) << second;
39
40 } // end function printUniversal
41

```

[Outline](#)

time1.cpp (2 of 3)

32


```

42 // print Time in standard format
43 void Time::printStandard()
44 {
45     cout << ( ( hour == 0 || hour == 12 ) ? 12 : hour % 12 )
46         << ":" << setfill( '0' ) << setw( 2 ) << minute
47         << ":" << setw( 2 ) << second
48         << ( hour < 12 ? " AM" : " PM" );
49
50 } // end function printStandard

```


[Outline](#)

33

time1.cpp (3 of 3)

```

1 // Fig. 6.7: fig06_07.cpp
2 // Program to test class Time.
3 // NOTE: This file must be compiled with time1.cpp.
4 #include <iostream>
5
6 using std::cout;
7 using std::endl;
8
9 // include definition of class Time from time1.h
10 #include "time1.h"
11
12 int main()
13 {
14     Time t; // instantiate object t of class Time
15
16     // output Time object t's initial values
17     cout << "The initial universal time is ";
18     t.printUniversal(); // 00:00:00
19     cout << "\nThe initial standard time is ";
20     t.printStandard(); // 12:00:00 AM
21
22     t.setTime( 13, 27, 6 ); // change time
23

```


[Outline](#)

34

fig06_07.cpp
(1 of 2)

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



```

24 // output Time object t's new values
25 cout << "\n\nUniversal time after setTime is ";
26 t.printUniversal(); // 13:27:06
27 cout << "\n\nStandard time after setTime is ";
28 t.printStandard(); // 1:27:06 PM
29
30 t.setTime( 99, 99, 99 ); // attempt invalid settings
31
32 // output t's values after specifying invalid values
33 cout << "\n\nAfter attempting invalid settings:"
34     << "\n\nUniversal time: ";
35 t.printUniversal(); // 00:00:00
36 cout << "\n\nStandard time: ";
37 t.printStandard(); // 12:00:00 AM
38 cout << endl;
39
40 return 0;
41
42 } // 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

Outline

fig06_07.cpp
(2 of 2)

fig06_07.cpp
output (1 of 1)

35

Controlling Access to Members



- Access modes
 - **private**
 - Default access mode
 - Accessible to member functions and **friends**
 - **public**
 - Accessible to any function in program with handle to class object
 - **protected**
 - later

36

```

1 // Fig. 6.8: fig06_08.cpp
2 // Demonstrate errors resulting from attempts
3 // to access private class members.
4 #include <iostream>
5
6 using std::cout;
7
8 // include definition of class Time from time1.h
9 #include "time1.h"
10
11 int main()
12 {
13     Time t; // create Time object
14
15     t.hour = 7; // error: 'Time::hour' is not accessible
16
17     // error: 'Time::minute' is not accessible
18     cout << "minute = " << t.minute;
19
20     return 0;
21 } // end main

```

Outline

fig06_08.cpp
(1 of 1)

37

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.

Controlling Access to Members

38

- 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

39

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

```

1  // Fig. 6.9: salesp.h
2  // SalesPerson class definition.
3  // Member functions defined in salesp.cpp.
4  #ifndef SALESP_H
5  #define SALESP_H
6
7  class SalesPerson {
8
9  public:
10     SalesPerson();           // constructor
11     void getSalesFromUser(); // input sales from keyboard
12     void setSales( int, double ); // set sales for a month
13     void printAnnualSales(); // summarize and print sales
14
15 private:
16     double totalAnnualSales(); // utility function
17     double sales[ 12 ];        // 12 monthly sales figures
18
19 }; // end class SalesPerson
20
21 #endif

```

[Outline](#)
 salesp.h (1 of 1)

Set access function performs validity checks.

private utility function.

```

1 // Fig. 6.10: salesp.cpp
2 // Member functions for class SalesPerson.
3 #include <iostream>
4
5 using std::cout;
6 using std::cin;
7 using std::endl;
8 using std::fixed;
9
10 #include <iomanip>
11
12 using std::setprecision;
13
14 // include SalesPerson class definition from salesp.h
15 #include "salesp.h"
16
17 // initialize elements of array sales to 0.0
18 SalesPerson::SalesPerson()
19 {
20     for ( int i = 0; i < 12; i++ )
21         sales[ i ] = 0.0;
22
23 } // end SalesPerson constructor
24

```

Outline

salesp.cpp (1 of 3)

41

```

25 // get 12 sales figures from the user at the keyboard
26 void SalesPerson::getSalesFromUser()
27 {
28     double salesFigure;
29
30     for ( int i = 1; i <= 12; i++ ) {
31         cout << "Enter sales amount for month " << i << ": ";
32         cin >> salesFigure;
33         setSales( i, salesFigure );
34     } // end for
35 } // end function getSalesFromUser
36
37 // set one of the 12 monthly sales figures; function subtracts
38 // one from month value for proper subscript in sales array
39 void SalesPerson::setSales( int month, double amount )
40 {
41     // test for valid month and amount values
42     if ( month >= 1 && month <= 12 && amount > 0 )
43         sales[ month - 1 ] = amount; // adjust for subscripts 0-11
44
45     else // invalid month or amount value
46         cout << "Invalid month or sales figure" << endl;
47 }
48

```

Outline

salesp.cpp (2 of 3)

42

Set access function performs validity checks.

```

49
50 } // end function setSales
51
52 // print total annual sales (with help of utility function)
53 void SalesPerson::printAnnualSales()
54 {
55     cout << setprecision( 2 ) << fixed
56         << "\nThe total annual sales are: $"
57         << totalAnnualSales() << endl; // call utility function
58 } // end function printAnnualSales
59
60 // private utility function to total annual sales
61 double SalesPerson::totalAnnualSales()
62 {
63     double total = 0.0;           // initialize total
64
65     for ( int i = 0; i < 12; i++ ) // summarize sales results
66         total += sales[ i ];
67
68     return total;
69 }
70
71 } // end function totalAnnualSales

```

[Outline](#)

salesp.cpp (3 of 3)

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

```

1 // Fig. 6.11: fig06_11.cpp
2 // Demonstrating a utility function.
3 // Compile this program with salesp.cpp
4
5 // include SalesPerson class definition from salesp.h
6 #include "salesp.h"
7
8 int main()
9 {
10     SalesPerson s;           // create SalesPerson object s
11
12     s.getSalesFromUser(); // note simple sequential code; no
13     s.printAnnualSales(); // control structures in main
14
15     return 0;
16 } // end main

```

[Outline](#)

fig06_11.cpp
(1 of 1)

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

```

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

```

Outline

fig06_11.cpp
output (1 of 1)

45

Initializing Class Objects: Constructors

46

- 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, ... );
```

Using Default Arguments with Constructors

47

• 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

```

1 // Fig. 6.12: time2.h
2 // Declaration of class Time.
3 // Member functions defined in time2.cpp.
4
5 // prevent multiple inclusions of header file
6 #ifndef TIME2_H
7 #define TIME2_H
8
9 // Time abstract data type definition
10 class Time {
11
12 public:
13     Time( int = 0, int = 0, int = 0 ); // default constructor
14     void setTime( int, int, int ); // set hour, minute, second
15     void printUniversal();          // print universal-time format
16     void printStandard();          // print standard-time format
17
18 private:
19     int hour;    // 0 - 23 (24-hour clock format)
20     int minute;  // 0 - 59
21     int second;  // 0 - 59
22
23 }; // end class Time
24
25 #endif

```

Default constructor specifying all arguments.



Outline

48

time2.h (1 of 1)


```

1 // Fig. 6.13: time2.cpp
2 // Member-function definitions for class Time.
3 #include <iostream>
4
5 using std::cout;
6
7 #include <iomanip>
8
9 using std::setfill;
10 using std::setw;
11
12 // include definition of class Time from time2.h
13 #include "time2.h"
14
15 // Time constructor initializes each data member to zero;
16 // ensures all Time objects start in a consistent state
17 Time::Time( int hr, int min, int sec )
18 {
19     setTime( hr, min, sec ); // validate and set time
20 }
21 // end Time constructor
22

```

Outline

time2.cpp (1 of 3)

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

49

```

23 // set new Time value using universal time, perform validity
24 // checks on the data values and set invalid values to zero
25 void Time::setTime( int h, int m, int s )
26 {
27     hour = ( h >= 0 && h < 24 ) ? h : 0;
28     minute = ( m >= 0 && m < 60 ) ? m : 0;
29     second = ( s >= 0 && s < 60 ) ? s : 0;
30
31 } // end function setTime
32
33 // print Time in universal format
34 void Time::printUniversal()
35 {
36     cout << setfill( '0' ) << setw( 2 ) << hour << ":"
37         << setw( 2 ) << minute << ":"
38         << setw( 2 ) << second;
39 }
40 // end function printUniversal
41

```

Outline

time2.cpp (2 of 3)

50

51





[Outline](#)

fig06_14.cpp
(1 of 2)

```

1  // Fig. 6.14: fig06_14.cpp
2  // Demonstrating a default constructor for class Time.
3  #include <iostream>
4
5  using std::cout;
6  using std::endl;
7
8  // include definition of class Time from time2.h
9  #include "time2.h"
10
11 int main()
12 {
13     Time t1;           // all arguments defaulted
14     Time t2( 2 );      // minute and second defaulted
15     Time t3( 21, 34 ); // second defaulted
16     Time t4( 12, 25, 42 ); // all values specified
17     Time t5( 27, 74, 99 ); // all bad values specified
18
19     cout << "Constructed with:\n\n"
20           << "all default arguments:\n ";
21     t1.printUniversal(); // 00:00:00
22     cout << "\n ";
23     t1.printStandard(); // 12:00:00 AM
24

```

Initialize **Time** objects using default arguments.

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

52





[Outline](#)

fig06_14.cpp
(2 of 2)

```

25     cout << "\n\nhour specified; default minute and second:\n ";
26     t2.printUniversal(); // 02:00:00
27     cout << "\n ";
28     t2.printStandard(); // 2:00:00 AM
29
30     cout << "\n\nhour and minute specified; default second:\n ";
31     t3.printUniversal(); // 21:34:00
32     cout << "\n ";
33     t3.printStandard(); // 9:34:00 PM
34
35     cout << "\n\nhour, minute, and second specified:\n ";
36     t4.printUniversal(); // 12:25:42
37     cout << "\n ";
38     t4.printStandard(); // 12:25:42 PM
39
40     cout << "\n\nall invalid values specified:\n ";
41     t5.printUniversal(); // 00:00:00
42     cout << "\n ";
43     t5.printStandard(); // 12:00:00 AM
44     cout << endl;
45
46     return 0;
47
48 } // end main

```

t5 constructed with invalid arguments; values set to 0.

53

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”

54

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

55

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**

56

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**

57
[Outline](#)
create.h (1 of 1)

```

1 // Fig. 6.15: create.h
2 // Definition of class CreateAndDestroy.
3 // Member functions defined in create.cpp.
4 #ifndef CREATE_H
5 #define CREATE_H
6
7 class CreateAndDestroy {
8 public:
9     CreateAndDestroy( int, char * ); // constructor
10    ~CreateAndDestroy(); // destructor
11
12 private:
13     int objectID;
14     char *message;
15
16 }; // end class CreateAndDestroy
17
18
19 #endif

```

Constructor and destructor member functions.

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

58
[Outline](#)
create.cpp (1 of 2)

```

1 // Fig. 6.16: create.cpp
2 // Member-function definitions for class CreateAndDestroy
3 #include <iostream>
4
5 using std::cout;
6 using std::endl;
7
8 // include CreateAndDestroy class definition from create.h
9 #include "create.h"
10
11 // constructor
12 CreateAndDestroy::CreateAndDestroy(
13     int objectNumber, char *messagePtr )
14 {
15     objectID = objectNumber;
16     message = messagePtr;
17
18     cout << "Object " << objectID << " constructor runs "
19         << message << endl;
20 } // end CreateAndDestroy constructor
21
22

```

Output message to demonstrate timing of constructor function calls.

59

▲

▼

Outline

create.cpp (2 of 2)

```
23 // destructor
24 CreateAndDestroy::~CreateAndDestroy()
25 {
26     // the following line is for pedagogic purposes only
27     cout << ( objectID == 1 || objectID == 6 ? "\n" : "" );
28
29     cout << "Object " << objectID << "   destructor runs   "
30           << message << endl;
31
32 } // end ~CreateAndDestroy destructor
```

Output message to demonstrate timing of destructor function calls.

60

▲

▼

Outline

fig06_17.cpp (1 of 3)

```
1 // Fig. 6.17: fig06_17.cpp
2 // Demonstrating the order in which constructors and
3 // destructors are called.
4 #include <iostream>
5
6 using std::cout;
7 using std::endl;
8
9 // include CreateAndDestroy class definition from create.h
10 #include "create.h"
11
12 void create( void ); // prototype
13
14 // global object
15 CreateAndDestroy first( 1, "(global before main)" );
16
17 int main()
18 {
19     cout << "\nMAIN FUNCTION: EXECUTION BEGINS" << endl;
20
21     CreateAndDestroy second( 2, "(local automatic in main)" );
22
23     static CreateAndDestroy third(
24         3, "(local static in main)" );
25 }
```

Create variable with global scope.

Create local automatic object.

Create static local object.

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

27

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

29

30 CreateAndDestroy fourth(4, "(local automatic in main)");

31

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

33

34 return 0;

35

36 } // end main

37

38 // function to create objects

39 void create(void)

40 {

41 cout << "\nCREATE FUNCTION: EXECUTION BEGINS" << endl;

42

43 CreateAndDestroy fifth(5, "(local automatic in create)");

44

45 static CreateAndDestroy sixth(

46 6, "(local static in create)");

47

48 CreateAndDestroy seventh(

49 7, "(local automatic in create)");

50

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.

Outline

fig06_17.cpp
(2 of 3)

61

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

52

53 } // end function create

fig06_17.cpp
(3 of 3)

62





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


```

1  // Fig. 6.18: time3.h
2  // Declaration of class Time.
3  // Member functions defined in time3.cpp
4
5  // prevent multiple inclusions of header file
6  #ifndef TIME3_H
7  #define TIME3_H
8
9  class Time {
10
11 public:
12     Time( int = 0, int = 0, int = 0 ); // default constructor
13
14     // set functions
15     void setTime( int, int, int ); // set hour, minute, second
16     void setHour( int ); // set hour
17     void setMinute( int ); // set minute
18     void setSecond( int ); // set second
19
20     // get functions
21     int getHour(); // return hour
22     int getMinute(); // return minute
23     int getSecond(); // return second
24

```

Outline

time3.h (1 of 2)

65



Set functions.

Get functions.

```

25     void printUniversal(); // output universal-time format
26     void printStandard(); // output standard-time format
27
28 private:
29     int hour; // 0 - 23 (24-hour clock format)
30     int minute; // 0 - 59
31     int second; // 0 - 59
32
33 }; // end clas Time
34
35 #endif

```

Outline

time3.h (2 of 2)

66

```

1 // Fig. 6.19: time3.cpp
2 // Member-function definitions for Time class.
3 #include <iostream>
4
5 using std::cout;
6
7 #include <iomanip>
8
9 using std::setfill;
10 using std::setw;
11
12 // include definition of class Time from time3.h
13 #include "time3.h"
14
15 // constructor function to initialize private data;
16 // calls member function setTime to set variables;
17 // default values are 0 (see class definition)
18 Time::Time( int hr, int min, int sec )
19 {
20     setTime( hr, min, sec );
21 }
22 // end Time constructor
23

```

Outline

time3.cpp (1 of 4)

67

```

24 // set hour, minute and second values
25 void Time::setTime( int h, int m, int s )
26 {
27     setHour( h );
28     setMinute( m );
29     setSecond( s );
30 } // end function setTime
31
32 // set hour value
33 void Time::setHour( int h )
34 {
35     hour = ( h >= 0 && h < 24 ) ? h : 0;
36 } // end function setHour
37
38 // set minute value
39 void Time::setMinute( int m )
40 {
41     minute = ( m >= 0 && m < 60 ) ? m : 0;
42 } // end function setMinute
43
44
45
46

```

Outline

time3.cpp (2 of 4)

68

Call set functions to perform validity checking.

Set functions perform validity checks before modifying data.

```

47 // set second value
48 void Time::setSecond( int s )
49 {
50     second = ( s >= 0 && s < 60 ) ? s : 0;
51 }
52 // end function setSecond
53
54 // return hour value
55 int Time::getHour()
56 {
57     return hour;
58 }
59 // end function getHour
60
61 // return minute value
62 int Time::getMinute()
63 {
64     return minute;
65 }
66 // end function getMinute
67

```

Set function performs validity checks before modifying data.

69

[Outline](#)

time3.cpp (3 of 4)

Get functions allow client to read data.

```

68 // return second value
69 int Time::getSecond()
70 {
71     return second;
72 }
73 // end function getSecond
74
75 // print Time in universal format
76 void Time::printUniversal()
77 {
78     cout << setfill( '0' ) << setw( 2 ) << hour << ":"
79         << setw( 2 ) << minute << ":"
80         << setw( 2 ) << second;
81 }
82 // end function printUniversal
83
84 // print Time in standard format
85 void Time::printStandard()
86 {
87     cout << ( ( hour == 0 || hour == 12 ) ? 12 : hour % 12 )
88         << ":" << setfill( '0' ) << setw( 2 ) << minute
89         << ":" << setw( 2 ) << second
90         << ( hour < 12 ? " AM" : " PM" );
91 }
92 // end function printStandard

```

Get function allows client to read data.

70

[Outline](#)

time3.cpp (4 of 4)

71





[Outline](#)

fig06_20.cpp
(1 of 3)

```

1  // Fig. 6.20: fig06_20.cpp
2  // Demonstrating the Time class set and get functions
3  #include <iostream>
4
5  using std::cout;
6  using std::endl;
7
8  // include definition of class Time from time3.h
9  #include "time3.h"
10
11 void incrementMinutes( Time &, const int ); // prototype
12
13 int main()
14 {
15     Time t;           // create Time object
16
17     // set time using individual set functions
18     t.setHour( 17 );   // set hour to valid value
19     t.setMinute( 34 ); // set minute to valid value
20     t.setSecond( 25 ); // set second to valid value
21

```

Invoke set functions to set valid values.

72





[Outline](#)

fig06_20.cpp
(2 of 3)

```

22 // use get functions to obtain hour, minute and second
23 cout << "Result of setting all valid values:\n"
24     << "  Hour: " << t.getHour()
25     << " Minute: " << t.getMinute()
26     << " Second: " << t.getSecond();
27
28 // set time using individual set functions
29 t.setHour( 234 ); // invalid hour set to 0
30 t.setMinute( 43 ); // set minute to valid value
31 t.setSecond( 6373 ); // invalid second set to 0
32
33 // display hour, minute and second after setting
34 // invalid hour and second values
35 cout << "\n\nResult of attempting to set invalid hour and"
36     << " second:\n  Hour: " << t.getHour()
37     << " Minute: " << t.getMinute()
38     << " Second: " << t.getSecond() << "\n\n";
39
40 t.setTime( 11, 58, 0 ); // set time
41 incrementMinutes( t, 3 ); // increment t's minute by 3
42
43 return 0;
44
45 } // end main
46

```

Attempt to set invalid values using set functions.

Invalid values result in setting data members to 0.

Modify data members using function setTime.

73

Outline

fig06_20.cpp

47 // add specified number of minutes to a Time object

48 void incrementMinutes(Time &tt, const int count)

49 {

50 cout << "Incrementing minute " << count

51 << " times:\nStart time: ";

52 tt.printStandard();

53

54 for (int i = 0; i < count; i++) {

55 tt.setMinute((tt.getMinute() + 1) % 60);

56

57 if (tt.getMinute() == 0)

58 tt.setHour((tt.getHour() + 1) % 24);

59

60 cout << "\nminute + 1: ";

61 tt.printStandard();

62

63 } // end for

64

65 cout << endl;

66

67 } // end function incrementMinutes

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

74

Outline

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.

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

75

- Reference to object
 - `&pRef = p;`
 - 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

```

1 // Fig. 6.21: time4.h
2 // Declaration of class Time.
3 // Member functions defined in time4.cpp
4
5 // prevent multiple inclusions of header file
6 #ifndef TIME4_H
7 #define TIME4_H
8
9 class Time {
10
11 public:
12     Time( int = 0, int = 0, int = 0 );
13     void setTime( int, int, int );
14     int getHour();
15
16     int &badSetHour( int ); // DANGEROUS reference return
17
18 private:
19     int hour;
20     int minute;
21     int second;
22
23 }; // end class Time
24
25 #endif

```

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



Outline


76


time4.h (1 of 1)

```

1 // Fig. 6.22: time4.cpp
2 // Member-function definitions for Time class.
3
4 // include definition of class Time from time4.h
5 #include "time4.h"
6
7 // constructor function to initialize private data;
8 // calls member function setTime to set variables;
9 // default values are 0 (see class definition)
10 Time::Time( int hr, int min, int sec )
11 {
12     setTime( hr, min, sec );
13 }
14 // end Time constructor
15
16 // set values of hour, minute and second
17 void Time::setTime( int h, int m, int s )
18 {
19     hour = ( h >= 0 && h < 24 ) ? h : 0;
20     minute = ( m >= 0 && m < 60 ) ? m : 0;
21     second = ( s >= 0 && s < 60 ) ? s : 0;
22 }
23 // end function setTime
24

```





[Outline](#)


77


time4.cpp (1 of 2)

```

25 // return hour value
26 int Time::getHour()
27 {
28     return hour;
29 }
30 // end function getHour
31
32 // POOR PROGRAMMING PRACTICE:
33 // Returning a reference to a private data member.
34 int &Time::badSetHour( int hh )
35 {
36     hour = ( hh >= 0 && hh < 24 )
37
38     return hour; // DANGEROUS reference return
39
40 } // end function badSetHour

```



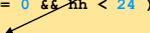


[Outline](#)

78

time4.cpp (2 of 2)

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



79

▲

▼

Outline

fig06_23.cpp
(1 of 2)

```
1 // Fig. 6.23: fig06_23.cpp
2 // Demonstrating a public member function that
3 // returns a reference to a private data member.
4 #include <iostream>
5
6 using std::cout;
7 using std::endl;
8
9 // include definition of class Time from time4.h
10 #include "time4.h"
11
12 int main()
13 {
14     Time t;
15
16     // store in hourRef the reference returned by badSetHour
17     int &hourRef = t.badSetHour( 20 );
18
19     cout << "Hour before modification: " << t.getHour();
20
21     // use hourRef to set invalid value
22     hourRef = 30;
23
24     cout << "\nHour after modification: " << t.getHour();
25 }
```

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

Reference allows setting of
private data member
hour.

80

▲

▼

Outline

fig06_23.cpp
(2 of 2)

fig06_23.cpp
output (1 of 1)

```
26 // Dangerous: Function call that returns
27 // a reference can be used as an lvalue!
28 t.badSetHour( 12 ) = 74;
29
30 cout << "\n\n*****" << endl;
31 << "POOR PROGRAMMING PRACTICE!!!!!!!" << endl;
32 << "badSetHour as an lvalue, Hour: 74" << endl;
33 << t.getHour() << endl;
34 << "\n*****" << endl;
35
36 return 0;
37
38 } // 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**.

81

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

```

1 // Fig. 6.24: fig06_24.cpp
2 // Demonstrating that class objects can be assigned
3 // to each other using default memberwise assignment.
4 #include <iostream>
5
6 using std::cout;
7 using std::endl;
8
9 // class Date definition
10 class Date {
11
12 public:
13     Date( int = 1, int = 1, int = 1990 ); // default constructor
14     void print();
15
16 private:
17     int month;
18     int day;
19     int year;
20
21 }; // end class Date
22

```


[Outline](#)

 fig06_24.cpp
 (1 of 3)

82

```

23 // Date constructor with no range checking
24 Date::Date( int m, int d, int y )
25 {
26     month = m;
27     day = d;
28     year = y;
29
30 } // end Date constructor
31
32 // print Date in the format mm-dd-yyyy
33 void Date::print()
34 {
35     cout << month << '-' << day << '-' << year;
36 } // end function print
37
38
39 int main()
40 {
41     Date date1( 7, 4, 2002 );
42     Date date2; // date2 defaults to 1/1/1990
43

```

[Outline](#)

fig06_24.cpp
 (2 of 3)

83

```

44     cout << "date1 = ";
45     date1.print();
46     cout << "\ndate2 = ";
47     date2.print();
48
49     date2 = date1; // default memberwise assignment
50
51     cout << "\n\nAfter default memberwise assignment, date2 = ";
52     date2.print();
53     cout << endl;
54
55     return 0;
56
57 } // end main

```

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)

84

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

Software Reusability

85

- 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

`const` (Constant) Objects and `const` Member Functions

86

- Principle of least privilege
 - Only allow modification of necessary objects
- Keyword **`const`**
 - Specify object not modifiable
 - Compiler error if attempt to modify **`const`** object
 - Example


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

 - Declares **`const`** object **`noon`** of class **`Time`**
 - Initializes to 12

`const` (Constant) Objects and `const` Member Functions

87

- **`const`** member functions
 - Member functions for **`const`** objects must also be **`const`**
 - Cannot modify object
 - Specify **`const`** in both prototype and definition
 - Prototype
 - After parameter list
 - Definition
 - Before beginning left brace

`const` (Constant) Objects and `const` Member Functions

88

- Constructors and destructors
 - Cannot be **`const`**
 - Must be able to modify objects
 - Constructor
 - Initializes objects
 - Destructor
 - Performs termination housekeeping

```

1 // Fig. 7.1: time5.h
2 // Definition of class Time.
3 // Member functions defined in time5.cpp.
4 #ifndef TIME5_H
5 #define TIME5_H
6
7 class Time {
8
9 public:
10     Time( int = 0, int = 0, int = 0 ); // default constructor
11
12     // set functions
13     void setTime( int, int, int ); // set time
14     void setHour( int );           // set hour
15     void setMinute( int );         // set minute
16     void setSecond( int );         // set second
17
18     // get functions (normally declared const)
19     int getHour() const;           // return hour
20     int getMinute() const;         // return minute
21     int getSecond() const;         // return second
22
23     // print functions (normally declared const)
24     void printUniversal() const;   // print universal time
25     void printStandard();          // print standard time

```

[Outline](#)
 time5.h (1 of 2)

Declare **const** get functions.

Declare **const** function **printUniversal**.

```

26
27 private:
28     int hour;    // 0 - 23 (24-hour clock format)
29     int minute;  // 0 - 59
30     int second;  // 0 - 59
31
32 }; // end class Time
33
34 #endif



```

[Outline](#)
 time5.h (2 of 2)

```

1 // Fig. 7.2: time5.cpp
2 // Member-function definitions for class Time.
3 #include <iostream>
4
5 using std::cout;
6
7 #include <iomanip>
8
9 using std::setfill;
10 using std::setw;
11
12 // include definition of class Time from time5.h
13 #include "time5.h"
14
15 // constructor function to initialize private data;
16 // calls member function setTime to set variables;
17 // default values are 0 (see class definition)
18 Time::Time( int hour, int minute, int second )
19 {
20     setTime( hour, minute, second );
21 }
22 // end Time constructor
23

```

[Outline](#)



time5.cpp (1 of 4)

91

```

24 // set hour, minute and second values
25 void Time::setTime( int hour, int minute, int second )
26 {
27     setHour( hour );
28     setMinute( minute );
29     setSecond( second );
30 }
31 // end function setTime
32
33 // set hour value
34 void Time::setHour( int h )
35 {
36     hour = ( h >= 0 && h < 24 ) ? h : 0;
37 }
38 // end function setHour
39
40 // set minute value
41 void Time::setMinute( int m )
42 {
43     minute = ( m >= 0 && m < 60 ) ? m : 0;
44 }
45 // end function setMinute
46

```

[Outline](#)

time5.cpp (2 of 4)

92

```

47 // set second value
48 void Time::setSecond( int s )
49 {
50     second = ( s >= 0 && s < 60 ) ? s : 0;
51 }
52 // end function setSecond
53
54 // return hour value
55 int Time::getHour() const
56 {
57     return hour;
58 }
59 // end function getHour
60
61 // return minute value
62 int Time::getMinute() const
63 {
64     return minute;
65 }
66 // end function getMinute
67

```

[Outline](#)

time5.cpp (3 of 4)

93

const functions do not modify objects.

```

68 // return second value
69 int Time::getSecond() const
70 {
71     return second;
72 }
73 // end function getSecond
74
75 // print Time in universal format
76 void Time::printUniversal() const
77 {
78     cout << setfill( '0' ) << setw( 2 ) << hour << ":"
79         << setw( 2 ) << minute << ":"
80         << setw( 2 ) << second;
81 }
82 // end function printUniversal
83
84 // print Time in standard format
85 void Time::printStandard() // note lack of const declaration
86 {
87     cout << ( ( hour == 0 || hour == 12 ) ? 12 : hour % 12 )
88         << ":" << setfill( '0' ) << setw( 2 ) << minute
89         << ":" << setw( 2 ) << second
90         << ( hour < 12 ? " AM" : " PM" );
91 }
92 // end function printStandard

```

[Outline](#)

time5.cpp (4 of 4)

94

const functions do not modify objects.

95

▲

▼

Outline

fig07_03.cpp
(1 of 2)

```
1 // Fig. 7.3: fig07_03.cpp
2 // Attempting to access a const object with
3 // non-const member functions.
4
5 // include Time class definition from time5.h
6 #include "time5.h"
7
8 int main()
9 {
10     Time wakeUp( 6, 45, 0 ); // non-constant object
11     const Time noon( 12, 0, 0 ); // constant object
12 }
```

Declare **noon** a **const** object.

Note that non-**const** constructor can initialize **const** object.

96

▲

▼

Outline

fig07_03.cpp
(2 of 2)

```
14         // OBJECT      MEMBER FUNCTION
15     wakeUp.setHour( 18 ); // non-const  non-const
16
17     noon.setHour( 12 ); // const      non-const
18
19     wakeUp.getHour(); // non-const  const
20
21     noon.getMinute(); // const      const
22     noon.printUniversal(); // const  const
23
24     noon.printStandard(); // const  non-const
25
26     return 0;
```

Compile error

error C2662: 'void Time::setHour(int)' : cannot convert 'this' pointer
from 'const Time' to 'Time &'

error C2662: 'void Time::printStandard (int)' : cannot convert 'this'
pointer from 'const Time' to 'Time &'

const (Constant) Objects and const Member Functions

97

- Member initializer syntax
 - Initializing with member initializer syntax
 - Can be used for
 - All data members
 - Must be used for
 - **const** data members
 - Data members that are references

```

1  // Fig. 7.4: fig07_04.cpp
2  // Using a member initializer to initialize a
3  // constant of a built-in data type.
4  #include <iostream>
5
6  using std::cout;
7  using std::endl;
8
9  class Increment {
10
11 public:
12     Increment( int c = 0, int i = 1 ); // default constructor
13
14     void addIncrement()
15     {
16         count += increment;
17     } // end function addIncrement
18
19     void print() const; // prints count and increment
20
21 
```


[Outline](#)

 fig07_04.cpp
 (1 of 3)

98

99

Outline

fig07_04.cpp
(2 of 3)

```
22 private:
23     int count;
24     const int increment; // const data member
25
26 }; // end class Increment
27
28 // constructor
29 Increment::Increment( int c, int i )
30 : count( c ), // initializer for non-const member
31   increment( i ) // required initializer for const member
32 {
33     // empty body
34 } // end Increment constructor
35
36 // print count and increment values
37 void Increment::print() const
38 {
39     cout << "count = " << count
40         << ", increment = " << increment << endl;
41 } // end function print
42
43
44
```

Declare increment as **const** data member.

Member initializer list separated from parameter list by colon.

Member initializer syntax can be used for non-**const** data member **count**.

Member initializer syntax must be used for **const** data member **increment**.

Member initializer consists of data member name (**increment**) followed by parentheses containing initial value (**c**).

100

Outline

fig07_04.cpp
(3 of 3)

fig07_04.cpp
output (1 of 1)

```
45 int main()
46 {
47     Increment value( 10, 5 );
48
49     cout << "Before incrementing: ";
50     value.print();
51
52     for ( int j = 0; j < 3; j++ ) {
53         value.addIncrement();
54         cout << "After increment " << j + 1 << ": ";
55         value.print();
56     }
57
58     return 0;
59 }
60 // end main
```

```
Before incrementing: count = 10, increment = 5
After increment 1: count = 15, increment = 5
After increment 2: count = 20, increment = 5
After increment 3: count = 25, increment = 5
```

101

Composition: Objects as Members of Classes

- **Composition**

- Class has objects of other classes as members

- **Construction of objects**

- Member objects constructed in order declared
 - Not in order of constructor's member initializer list
 - Constructed before enclosing class objects (host objects)

102

```

1 // Fig. 7.6: date1.h
2 // Date class definition.
3 // Member functions defined in date1.cpp
4 #ifndef DATE1_H
5 #define DATE1_H
6
7 class Date {
8
9 public:
10     Date( int = 1, int = 1, int = 1900 ); // default constructor
11     void print() const; // print date in month/day/year format
12     ~Date(); // provided to confirm destruction order
13
14 private:
15     int month; // 1-12 (January-December)
16     int day; // 1-31 based on month
17     int year; // any year
18
19     // utility function to test proper day for month and year
20     int checkDay( int ) const;
21
22 }; // end class Date
23
24 #endif

```

Note no constructor with parameter of type **Date**. Recall compiler provides default copy constructor.


[Outline](#)

date1.h (1 of 1)

```

1 // Fig. 7.7: date1.cpp
2 // Member-function definitions for class Date.
3 #include <iostream>
4
5 using std::cout;
6 using std::endl;
7
8 // include Date class definition from date1.h
9 #include "date1.h"
10
11 // constructor confirms proper value for month; calls
12 // utility function checkDay to confirm proper value for day
13 Date::Date( int mn, int dy, int yr )
14 {
15     if ( mn > 0 && mn <= 12 ) // validate the month
16         month = mn;
17
18     else { // invalid month set to 1
19         month = 1;
20         cout << "Month " << mn << " invalid. Set to month 1.\n";
21     }
22
23     year = yr; // should validate yr
24     day = checkDay( dy ); // validate the day
25

```

[Outline](#)

date1.cpp (1 of 3)

103

```

1 // Fig. 7.8: employee1.h
2 // Employee class definition.
3 // Member functions defined in employee1.cpp.
4 #ifndef EMPLOYEE1_H
5 #define EMPLOYEE1_H
6
7 // include Date class definition from date1.h
8 #include "date1.h"
9
10 class Employee {
11
12 public:
13     Employee(
14         const char *, const char *, const Date &, const Date & );
15
16     void print() const;
17     ~Employee(); // provided to confirm destruction order
18
19 private:
20     char firstName[ 25 ];
21     char lastName[ 25 ];
22     const Date birthDate; // composition: member object
23     const Date hireDate; // composition: member object
24
25 }; // end class Employee

```

[Outline](#)

employee1.h (1 of 2)

104

Using composition;
Employee object contains
Date objects as data
 members.

```

26
27 #endif

1 // Fig. 7.9: employee1.cpp
2 // Member-function definitions for class Employee.
3 #include <iostream>
4
5 using std::cout;
6 using std::endl;
7
8 #include <cstring>      // strcpy and strlen prototypes
9
10 #include "employee1.h" // Employee class definition
11 #include "date1.h"     // Date class definition
12

```

105
[Outline](#)
employee1.h (2 of 2)
employee1.cpp
(1 of 3)

```

13 // constructor uses member initializer list to pass initializer
14 // values to constructors of member objects birthDate and
15 // hireDate [Note: This invokes the so-called "default copy
16 // constructor" which the C++ compiler provides implicitly.]
17 Employee::Employee( const char *first, const char *last,
18     const Date &dateOfBirth, const Date &dateOfHire )
19     : birthDate( dateOfBirth ), // initialize birthDate
20       hireDate( dateOfHire )    // initialize hireDate
21 {
22     // copy first into firstName and be sure
23     int length = strlen( first );
24     length = ( length < 25 ? length : 24 );
25     strncpy( firstName, first, length );
26     firstName[ length ] = '\0';
27
28     // copy last into lastName and be sure that it fits
29     length = strlen( last );
30     length = ( length < 25 ? length : 24 );
31     strncpy( lastName, last, length );
32     lastName[ length ] = '\0';
33
34     // output Employee object to show when constructor is called
35     cout << "Employee object constructor: "
36           << firstName << " " << lastName << endl;
37

```

106
[Outline](#)
employee1.cpp
(2 of 3)

Member initializer syntax to initialize **Date** data members **birthDate** and **hireDate**; compiler uses default copy constructor.

Output to show timing of constructors.

```

38 } // end Employee constructor
39
40 // print Employee object
41 void Employee::print() const
42 {
43     cout << lastName << ", " << firstName << "\nHired: ";
44     hireDate.print();
45     cout << " Birth date: ";
46     birthDate.print();
47     cout << endl;
48
49 } // end function print
50
51 // output Employee object to show when it
52 Employee::~Employee()
53 {
54     cout << "Employee object destructor: "
55           << lastName << ", " << firstName << endl;
56
57 } // end destructor ~Employee

```

Outline

employee1.cpp
(3 of 3)

107

Output to show timing of destructors.

```

1 // Fig. 7.10: fig07_10.cpp
2 // Demonstrating composition--an object with member objects.
3 #include <iostream>
4
5 using std::cout;
6 using std::endl;
7
8 #include "employee1.h" // Employee class definition
9
10 int main()
11 {
12     Date birth( 7, 24, 1949 );
13     Date hire( 3, 12, 1988 );
14     Employee manager( "Bob", "Jones", birth, hire );
15
16     cout << '\n';
17     manager.print();
18
19     cout << "\nTest Date constructor with invalid values:\n";
20     Date lastDayOff( 14, 35, 1994 ); // invalid month and day
21     cout << endl;
22
23     return 0;
24
25 } // end main

```

Outline

fig07_10.cpp
(1 of 1)

108

Create Date objects to pass to Employee constructor.

109

Outline

10.cpp (1 of 1)

```

Date object constructor for date 7/24/1949
Date object constructor for date 3/12/1988
Employee object constructor: Bob Jones

Jones, Bob
Hired: 3/12/1988 Birth date: 7/24/1949

Test Date constructor with invalid values:
Month 14 invalid. Set to month 1.
Day 35 invalid. Set to day 1.
Date object constructor for date 1/1/1994

Date object destructor for date 1/1/1994
Employee object destructor: Jones, Bob
Date object destructor for date 3/12/1988
Date object destructor for date 7/24/1949
Date object destructor for date 3/12/1988
Date object destructor for date 7/24/1949

```

Note two additional **Date** objects constructed; no output since default copy constructor used.

Destructor for host object **manager** runs before destructors for member objects **hireDate** and **birthDate**.

Destructor for **Employee**'s member object **hireDate**.

Destructor for **Employee**'s member object **birthDate**.

Destructor for **Date** object **hire**.

Destructor for **Date** object **birth**.

110

Practice Time: Lab4

1. Download the file in E3
Lab004.zip → the same file in page 65 - 74
2. Add the three file in one project
compile and run
3. Add two file "Date.cpp", "Date.h"
and new Class "Date"
- 4.

Practice Time: Lab4

111

- 4. Date Class,
member Object:
 Year, Month, Day,
member function:
 Set Birthday[10]
 just like the example at page 40-44
 Sorting Birthday
 large to small

End
