Structure Definitions

- **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

- 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;`

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 .

© 2003 Prentice Hall, Inc. All rights reserved.
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

Define structure type Time with three integer members.
Pass references to constant Time objects to eliminate copying overhead.
Class

- Classes (keyword `class`)
  - Model objects
    - Attributes (data members)
    - Behaviors (member functions)
      - Methods
      - Invoked in response to messages
- Member access specifiers: `public`, `private`, `protected`:
- Constructor function
  - Special member function
    - Initializes data members
    - Same name as class
  - Called when object instantiated
  - Several constructors
    - Function overloading
    - No return type

Implementing a Time Abstract Data Type with a class

- Objects of class
  - After class definition
    - Class name new type specifier
    - Object, array, pointer and reference declarations
  - Member functions defined outside class
    - Binary scope resolution (`:`)
      ```
      ReturnType ClassName::MemberFunctionName(
      ) {...}
      ```
  - Member functions defined inside class
    - Do not need scope resolution operator, class name
    - Compiler attempts inline
      - Outside class, inline explicitly with keyword `inline`

```
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
```
// 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
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 member to zero and ensures all Time objects start in a consistent state
Time::Time()
{
    hour = minute = second = 0;
} // end Time constructor
// set new Time value using universal time, 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.

Define class Time.

public member function checks parameter values for validity before setting private data members.
47 // print Time in universal format
48 void Time::printUniversal()
49 {
50   cout << setfill( '0' ) << setw( 2 ) << hour << ":" << setw( 2 ) << minute << "":" << setw( 2 ) << second;
51 } // end function printUniversal
52
53 // print Time in standard format
54 void Time::printStandard()
55 {
56   cout << ( ( hour == 0 || hour == 12 ) ? 12 : hour % 12 ) << ":" << setfill( '0' ) << setw( 2 ) << minute << ":" << setw( 2 ) << second << ( hour < 12 ? " AM" : " PM" );
57 } // end function printStandard
58
59 int main()
60 {
61   Time t;  // instantiate object t of class Time
62   // output Time object t's initial values
63   cout << "The initial universal time is ";
64   t.printUniversal();   // 00:00:00
65   cout << "The initial standard time is ";
66   t.printStandard();    // 12:00:00 AM
67   t.setTime( 13, 27, 6 );  // change time
68   // output Time object t's new Time
69   cout << "The universal time after setTime is ";
70   t.printUniversal();   // 13:27:06
71   cout << "The standard time after setTime is ";
72   t.printStandard();    // 1:27:06 PM
73   t.setTime( 99, 99, 99 );  // attempt invalid settings
74   // output t's values after specifying invalid values
75   cout << "After attempting invalid settings:
76   cout << "Universal time: ";
77   t.printUniversal();   // 00:00:00
78 } // 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.

Classes

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

- 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
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

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

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

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
  - 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
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)

- **Class member access**
  - Default `private`
  - Explicitly set to `private, public, protected`

- **struct** member access
  - Default `public`
  - Explicitly set to `private, public, protected`

Access Functions and Utility Functions

- **Access to class’s `private` data**
  - Controlled with access functions (accessor methods)
    - Get function - Read `private` data
    - Set function - Modify `private` data

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

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

}; // end class SalesPerson

#endif

// get 12 sales figures from the user at the keyboard
void SalesPerson::getSalesFromUser()
{
  double salesFigure;
  
  for ( int i = 1; i <= 12; i++ ) {
    cout << "Enter sales amount for month " << i << " : ";
    cin >> salesFigure;
    setSales( i, salesFigure );
  } // end for

} // end function getSalesFromUser

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

} // end function setSales
Initializing Class Objects: Constructors

• Constructors
  – Initialize data members; no return type
    • Or can set later
  – Same name as class
  – Can specify default arguments
  – Default constructors
    • Defaults all arguments
    OR
    • Explicitly requires no arguments
    • Can be invoked with no arguments
    • Only one per class

• Initializers
  – Passed as arguments to constructor
  – In parentheses to right of class name before semicolon

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

Default constructor specifying all arguments.
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
}

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

// 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;
}

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

© 2003 Prentice Hall, Inc.
All rights reserved.
// 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 << "Construct with:

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

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

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

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

all invalid values specified:
  ";
t5.printUniversal();  // 00:00:00
    cout << 
  ";
t5.printStandard();   // 12:00:00 AM
    cout << endl;
    return 0;
} // end main
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”
- Constructors and destructors - Called implicitly by compiler
- Order of function calls
  - Depends on when execution enters and exits scope of objects
  - Generally, destructor calls reverse order of constructor calls

When Constructors and Destructors Are Called

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

void create( void ); // prototype

int main()
{
    cout << "MAIN FUNCTION: EXECUTION BEGINS" << endl;
    CreateAndDestroy second( 2, "(local automatic in main)" );
    static CreateAndDestroy third( 3, "(local static in main)" );
    create(); // call function to create objects
    cout << "MAIN FUNCTION: EXECUTION RESUMES" << endl;
    CreateAndDestroy fourth( 4, "(local automatic in main)" );
    cout << "MAIN FUNCTION: EXECUTION ENDS" << endl;
    return 0;
} // end main

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

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

void create( void ); // prototype

// global object
CreateAndDestroy first( 1, "(global before main)" );
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
Subtle Trap: Returning a Reference to a private Data Member

- 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

---

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

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

// 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 badSetHour
    int &hourRef = t.badSetHour( 20 );
    cout << "Hour before modification: " << hourRef;
    // use hourRef to set invalid
    hourRef = 30;
    cout << 
    %0d
    "Hour after modification: " << t.getHour();
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***************
" << "POOR PROGRAMMING PRACTICE:!!
" << "badSetHour as an lvalue, Hour: 74
" << "***************" << endl;
31
32  return 0;
33
34 } // end main

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
// 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

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

- **Keyword const**
  - Specify object not modifiable
  - Compiler error if attempt to modify `const` object
  - Example
    ```cpp
    const Time noon(12, 0, 0);
    ```
    - Declares `const` object `noon` of class `Time`
    - Initializes to 12

- **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**

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

- **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
// Fig. 7.4: fig07_04.cpp
// Using a member initializer to initialize a
// constant of a built-in data type.
#include <iostream>
using std::cout;
using std::endl;
class Increment {
  public:
    Increment( int c = 0, int i = 1 );  // default constructor
    void addIncrement()
    {
      count += increment;
    } // end function addIncrement
    void print() const;   // prints count and increment
  private:
    int count;
    const int increment;    // const data member
} // end class Increment

// constructor
Increment::Increment( int c, int i )
  : count( c ),    // initializer for non-const member
    increment( i )  // required initializer for const member
  {
    // empty body
  } // end Increment constructor

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

Declare increment as 
const data member.
Member initializer list
increment as const
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).
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)

// Fig. 7.6: date1.h
// Date class definition.
// Member functions defined in date1.cpp
#ifndef DATE1_H
#define DATE1_H

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

private:
  int month; // 1-12 (January-December)
  int day;    // 1-31 based on month
  int year;  // any year

  // utility function to test proper day for month and year
  int checkDay( int ) const;
}; // end class Date

#endif

Note no constructor with parameter of type Date. Recall compiler provides default copy constructor.
// Fig. 7.8: employee1.h
// Employee class definition.
// Member functions defined in employee1.cpp.
#ifndef EMPLOYEE1_H
#define EMPLOYEE1_H

// include Date class definition from date1.h
#include "date1.h"

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

private:
    char firstName[ 25 ];
    char lastName[ 25 ];
    const Date birthDate;  // composition: member object
    const Date hireDate;   // composition: member object
}; // end class Employee

// constructor uses member initializer list to pass initializer
// values to constructors of member objects birthDate and hireDate [Note: This invokes the so-called "default copy constructor" which the C++ compiler provides implicitly.]
Employee::Employee( const char *first, const char *last, const Date dateOfBirth, const Date dateOfHire ) :
    birthDate( dateOfBirth ),  // initialize birthDate
    hireDate( dateOfHire )     // initialize hireDate
{
    // copy first into firstName and be sure that it fits
    int length = strlen( first );
    length = ( length < 25 ? length : 24 );
    strncpy( firstName, first, length );
    firstName[ length ] = '\0';

    // copy last into lastName and be sure that it fits
    length = strlen( last );
    length = ( length < 25 ? length : 24 );
    strncpy( lastName, last, length );
    lastName[ length ] = '\0';

    // output Employee object to show when constructor is called
    cout << "Employee object constructor: "
        << firstName << ' ' << lastName << endl;
}
// Fig. 7.10: fig07_10.cpp
// Demonstrating composition—an object with member objects.
#include <iostream>
using std::cout;
using std::endl;
#include "employee.h" // Employee class definition
int main()
{
    // Create Date objects to pass to Employee constructor.
    Date birth(7, 24, 1949);
    Date hire(3, 12, 1988);
    Employee manager("Bob", "Jones", birth, hire);
    cout << "\n" << manager.print();
    cout << "\nTest Date constructor with invalid values:\n";
    Date lastDayOff(14, 35, 1994); // invalid month and day
    cout << endl;
    return 0;
} // end main