Ceng 205 Compter Programming II

Cem Özdoğan

29th July 2004

Contents

List of Tables

List of Figures

Chapter 1

Introduction, Classes and Data Abstraction

- Basic characteristics of O-O languages
  - Everything is an object.
  - Object-orientation is a natural way of thinking about the world and of writing computer programs.
  - Objects are all around us–people, animals, plants, cars, planes, buildings, computers, etc.
  - Abstractions allow us to view screen images as objects such as people, planes, trees, etc. rather than as individual dots of color.
  - Abstractions allow us to think in terms of beaches rather than grains of sand, houses rather than bricks.
  - All objects have attributes such as size, shape, color, weight, etc.
  - All objects exhibit various behaviors. A baby cries, sleeps, crawls, walks; a car accelerates, brakes, turns, etc.
  - Humans learn about objects by studying their attributes and observing their behaviors.
  - Different objects can have many of the same attributes and exhibit similar behaviors.
    - Comparisons can be made between babies and adults, and between humans and chimpanzees.
    - Cars, trucks, little red wagons, and roller blades have much in common.

  - Object-oriented programming (OOP) models real-world objects with software counterparts.
    - It takes advantage of class relationships where objects of a certain class, such as a class of vehicles, have the same characteristics.
    - It takes advantage of inheritance relationships, and even multiple inheritance relationships, where newly created classes are derived by inheriting characteristics of existing classes, yet contain unique characteristics of their own.
  - A program is a bunch of objects telling each other what to do, by sending messages.
  - Each object has its own memory, and is made up of other objects.
  - Every object has a type (class).
  - All objects of the same type can receive the same messages.

- Objects
  - An object has an **interface**, determined by the class it's an instance of.
  - A class is an **abstract data type** (or **user-defined data type**).
  - Defining a class requires defining its interface.
  - What about built-in types?
    - Think of an *int*
    - What's its interface?
    - How do you "send it messages"?
    - How do you make (construct) one?

- The interface is the critical part, but the details (implementation) are important too

- Users use the interface (the "public part"), the implementation is hidden by "access control".

- C libraries have always been like this, sort of:
  - The library designer invents a useful struct.
  - Then she provides some useful functions for the struct.
  - The user creates an instance of the struct, then applies library functions to it.

- C++ uses "access specifiers": **public, protected**, and **private** to determine who can use the attribute or function.
- Two Ways of Reusing Classes
  - **Composition:** One class has another as a "part".
  - **Inheritance:** One class is a specialized version of another
- **Polymorphism:** Different subclasses respond to the same message, possibly with different actions.
- Creating and Destroying Objects
  - We usually get this for free with built-in types like int or char, we just say
    - int i;
    - char c;
  - With user-defined types (the ones we make), we need to be explicit about what we want:
    - constructor function
    - destructor function
    - C++ has new and delete (similar to malloc and free in C)
    - This is a very important issue! What is a memory leak?
- A compiler typically does
  - preprocessing
  - first pass to make parse tree
  - second pass to generate code
- The result is an object module (.obj file).
- A linker produces an .exe file by
  - Resolving references between compilation units (i.e., separate source files)
  - Adding code from libraries
  - Adding special startup code
  - Building the final executable file

- In C++, variables and functions must be both declared and defined. The rules:
    - A declaration tells the compiler that you intend to use a variable/function with a certain name.
    - A variable declaration specifies the type (int, float, etc.) so the compiler can check your usage.
    - A variable declaration doesn't allocate space for the variable.
    - A function declaration specifies the function name, argument types, and return type, so the compiler can check your usage.
    - A function declaration doesn't allocate space for the function code.
    - A variable definition causes memory to be allocated to hold its value. This can only be done (must be done) exactly once in the entire program. Why?
    - And so for functions.
- Libraries are collections of compiled function definitions.
    - Library header files (.h files, or files with no extension) are collections of (uncompiled e.g., ASCII) function declarations.
    - *#include*ing a header file is a fast and painless way of providing the declarations the compiler insists on.
    - The compiler is happy, since it has declarations from the .h file(s)
    - The linker is happy, because there is exactly one definition of a library function.
    - The linker resolves references to variables/functions that are spread across files.
- Survey of Programming Techniques (see Fig. 1.1)
    - Unstructured programming.
        - Simple sequence of command statements.
        - Operates directly on global data.
        - Not good for large programs.
        - Repetitive statement segments are copied over.
        - The repetitive sequences extracted and named so that they can be called and values returned leads to the idea of procedures.

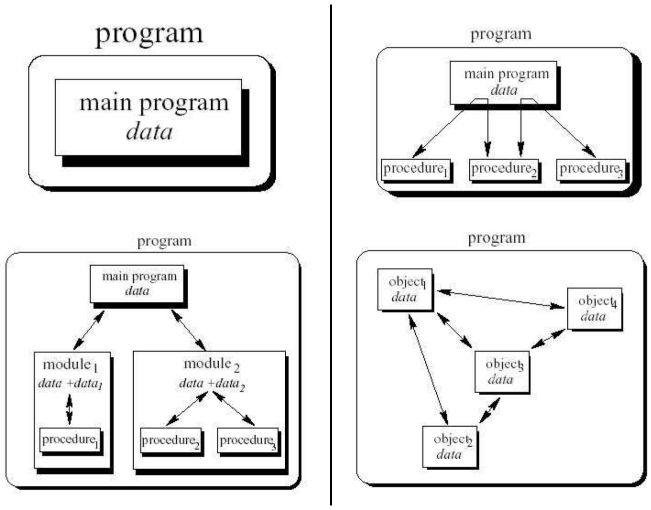


Figure 1.1: Survey of Programming Techniques; unstructured, procedural, modular, and object-oriented programming.

- Procedural programming.
    - ∗ Combines returning sequences of statements into one function.
    - ∗ Procedure calls are used to invoke procedures.
    - ∗ Programs are now more structured.
    - ∗ Errors are easier to detect.
    - ∗ Combining procedures into modules is the next logical extension.
- Modular programming.
    - ∗ Procedures with common functionality are grouped into modules.
    - ∗ Main program coordinates calls to procedures within modules.
    - ∗ Each module has its own data and isolated for other modules.
- Object-oriented programming.
    - ∗ Data and the functions that operate on that data are combined into an object.
    - ∗ Programming is not function based but object based.
    - ∗ Objects are base on three basic ideas: Encapsulation, Inheritance and Polymorphism.

1.1 History: The Rise and Decline of Structured Programming

For many years (roughly 1970 to 1990), *structured programming* was the most common way to organize a program. This is characterized by a functional-decomposition style - breaking the algorithms in to every smaller functions. This technique was a great improvement over the ad hoc programming which preceded it. However, as programs became larger, structured programming was not able control the exponential increase in complexity.

1.1.1 The Problem - Complexity

Complexity measurements grow exponentially as the size of programs grow. One measurement is *coupling*, or much different elements (modules, data structures) interact with each other. The fewer the connections, the less complex a program is. Low coupling is highly desirable.
There have been several post-structured programming attempts to control complexity. One of these is to use software *components* - preconstructed software "parts" to avoid programming. And when you have to program, use *object-oriented programming* (OOP).
Bjarne Stroustrup of Bell Labs extended the C language to be capable of Object-Oriented Programming (OOP), and it became popular in the 1990's as C++. There were several enhancements, but the central change was extending **struct** to allow it to contain functions and use inheritance. These extended structs were later renamed **classes.** A C++ standard was established in 1999, so there are variations in the exact dialect that is accepted by pre-standard compilers.

1.2 Object-Oriented Programming (OOP)

Object-Oriented Programming groups related data and functions together in a *class*, generally making data private and only some functions public. Restricting access decreases coupling and increases cohesion. While it is not a panacea, it has proven to be very effective in reducing the complexity increase with large programs. For small programs may be difficult to see the advantage of OOP over, eg, structured programming because there is little complexity regardless of how it's written. Many of the mechanics of OPP are easy to demonstrate; it is somewhat harder to create small, convincing examples.

OOP is often said to incorporate three techniques: inheritance, encapsulation, and polymorphism. Of these, you should first devote yourself to choosing the right classes (possibly difficult) and getting the encapsulation right (fairly easy). Inheritance and polymorphism are not even present in many programs, so you can ignore them at that start.

1.2.1 Encapsulation

Encapsulation is grouping data and functions together and keeping their implementation details private. Greatly restricting access to functions and data reduces *coupling*, which increases the ability to create large programs. Classes also encourage *coherence*, which means that a given class does one thing. By increasing coherence, a program becomes easier to understand, more simply organized, and this better organization is reflected in a further reduction in coupling.

1.2.2 Inheritance

Inheritance means that a new class can be defined in terms of an existing class. There are three common terminologies for the new class: the *derived* class, the *child* class, or the *subclass*. The original class is the *base* class, the *parent* class, or the *superclass*. The new child class inherits all capabilities of the parent class and adds its own fields and methods. Altho inheritance is very important, especially in many libraries, is often not used in an application.

1.2.3 Polymorphism

Polymorphism is the ability of different functions to be invoked with the same name. There are two forms.
Static polymorphism is the common case of *overriding* a function by providing additional definitions with different numbers or types of parameters. The compiler matches the parameter list to the appropriate function.
Dynamic polymorphism is much different and relies on parent classes to define *virtual functions* which child classes may redefine. When this virtual member function is called for an object of the parent class, the execution dynamically chooses the appropriate function to call - the parent function if the object really is the parent type, or the child function if the object really is the child type. This explanation is too brief to be useful without an example, but that will have to be written latter.

1.2.4 Advantages of OOP

- Re-use of code. Linking of code to objects and explicit specification of relations between objects allows related objects to share code. Encapsulation and weak coupling between objects means class definitions are more likely to be re-used in other applications. Objects as well as procedures (focus of C libraries) become likely candidates for re-use. The enforcement of a consistent interface to objects lessens code duplication.

- Ease of comprehension. Structure of code and data structures in it can be set up to closely mimic the generic application concepts and processes. High-level code could make some sense even to a non-programmer. The analysis/design/coding phases in development become more seamless since they can all deal in the same concepts.

- Ease of fabrication and maintenance (redesign and extension) facilitated by encapsulation, data abstraction which allow for very clean designs. When an object is going into disallowed states, only its methods need be investigated. This narrows down search for problems.

- C++ Objectives

  - extend C to allow for object-oriented programming
  - other improvements - some resulting in deprecation of some C facilities
  - remain compatible and comparable (syntax, performance, portability, design philosophy - don't pay for what you don't use, don't get stuck with things you don't need) with C
  - emphasize compile-time type checking

- C++ is multi-paradigm. It provides for the object-oriented approach but doesn't enforce its use. This makes it a good transition language and gives it flexibility when a particular situation doesn't fit the object-oriented philosophy.

- With this object-oriented approach, C++ overcomes certain shortcomings of C:

  - Lack of encapsulation means that if an object is getting trashed, it's difficult to find the code responsible. Many procedures may have had idiosyncratic interactions with the object.

  - Doesn't recognize relationships between types. Pointer casting necessary. In C++, pointer casting can just about always be dispensed with. Pointer casting is a kludge. Compiler can't check if you are doing it correctly. No type safety (see definition below).

  - Not easy to extend existing libraries; for example, make it so printf() can handle new types.

  - Except for FILEs, there are no well-developed objects (like stacks and lists) in the standard libraries.

- C's future is as a portable "universal" assembler, a back end for code generators.

- While any C++ compiler should be able to compile a C program successfully with minor changes, several aspects of C programming are discarded in the transition to C++: new facilities are supplied for I/O, memory allocation and error handling; macros and pointer casts become obsolete for the most part.

1.2.5 OOP Terminology

Along with each programming revolution comes a new set of terminology. There are some new OOP concepts, but many have a simple analog in pre-OOP practice.

OOP Term	Definition
method	Same as function, but the typical OO notation is used for the call, ie, f(x,y) is written x.f(y) where x is an object of class that contains this f method.
send a message	Call a function (method).
instantiate	Allocate a class/struct object (ie, instance) with new.
class	A struct with both data and functions.
object	Memory allocated to a class/struct. Often allocated with new.
member	A field or function is a member of a class if it's defined in that class
constructor	Function-like code that initializes new objects (structs) when they instantiated (allocated with new).
destructor	Function-like code that is called when an object is deleted to free any resources (eg, memory) that is has pointers to.
inheritance	Defining a class (child) in terms of another class (parent). All of the public members of the public class are available in the child class.
polymorphism	Defining functions with the same name, but different parameters.
overload	A function is overloaded if there is more than one definition. See polymorphism.
override	Redefine a function from a parent class in a child class.
subclass	Same as child, derived, or inherited class.
superclass	Same as parent or base class.
attribute	Same as data member or member field.

1.2.6 Other Object-Oriented Languages

- Objective C
- CLOS (Common Lisp Object System)
- Ada 9X
- FORTRAN 90
- Smalltalk
- Modula-3
- Eiffel

1.3 Structure Definitions

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

```
struct Time{ //structure tag
int hour;   //structure members
int minute; //structure members
int second; //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
- 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;
    ```

1.4 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;}
```

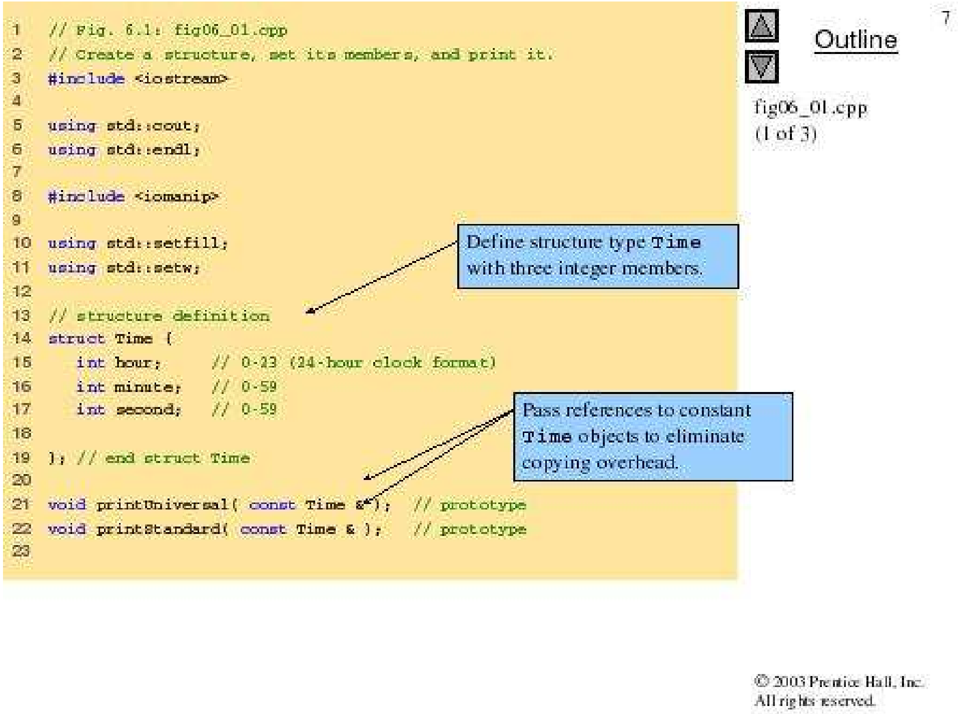
  - **timePtr**− >**hour** same as **(*timePtr).hour**
    - Parentheses required, ***** lower precedence than **.**

1.5 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

1.6 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



```
// 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

```

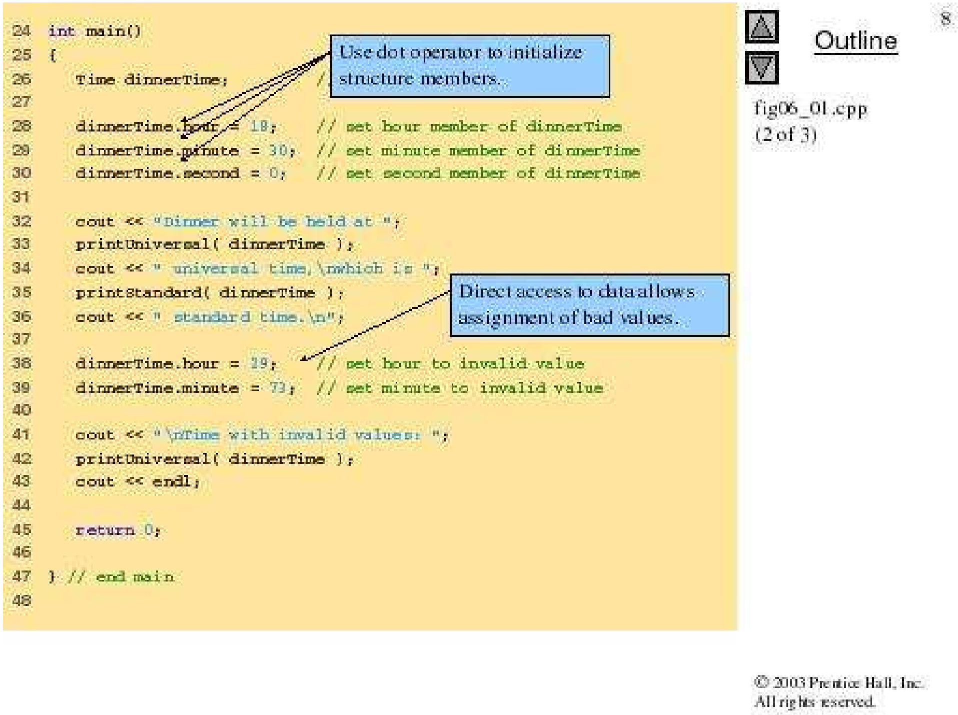
Define structure type `Time` with three integer members.

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

7

Outline

fig06_01.cpp
(1 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.

8

Outline

fig06_01.cpp
(2 of 3)

Figure 1.2: Creating a structure, setting its members and printing the structure (part 1 of 2).

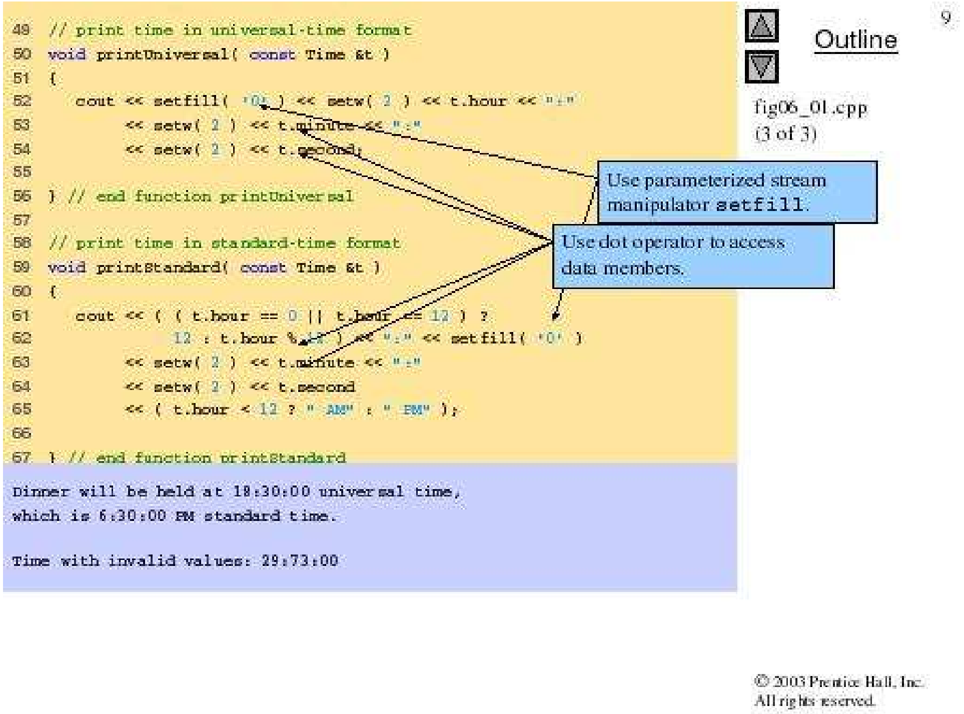


fig06_01.cpp
(3 of 3)

```
// 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
```

```
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
```

Use parameterized stream manipulator `setfill`.

Use dot operator to access data members.

Figure 1.3: Creating a structure, setting its members and printing the structure (part 2 of 2).

- **private:** Accessible only to member functions of class
- **protected:**

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

```
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
```

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

    ```
    Time sunset;
    Time arrayofTimes[ 5 ];
    Time *pointerToTime;
    Time \&dinnerTime = sunset;
    ```

- 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( ){
    .
    .
    .
    }
    ```

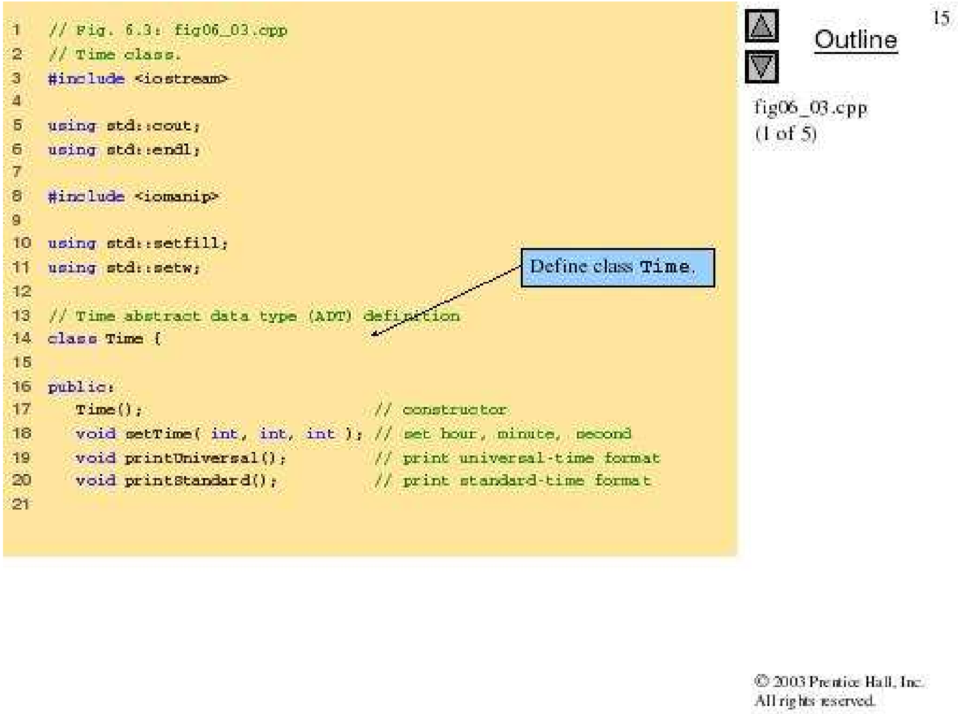
  - Does not change whether function **public** or **private**
- Member functions defined inside class
  - Do not need scope resolution operator, class name

  - Compiler attempts **inline**; Outside class, inline explicitly with keyword inline

- Destructor
  - 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

1.7 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
- Function scope
  - Variables declared in member function



15

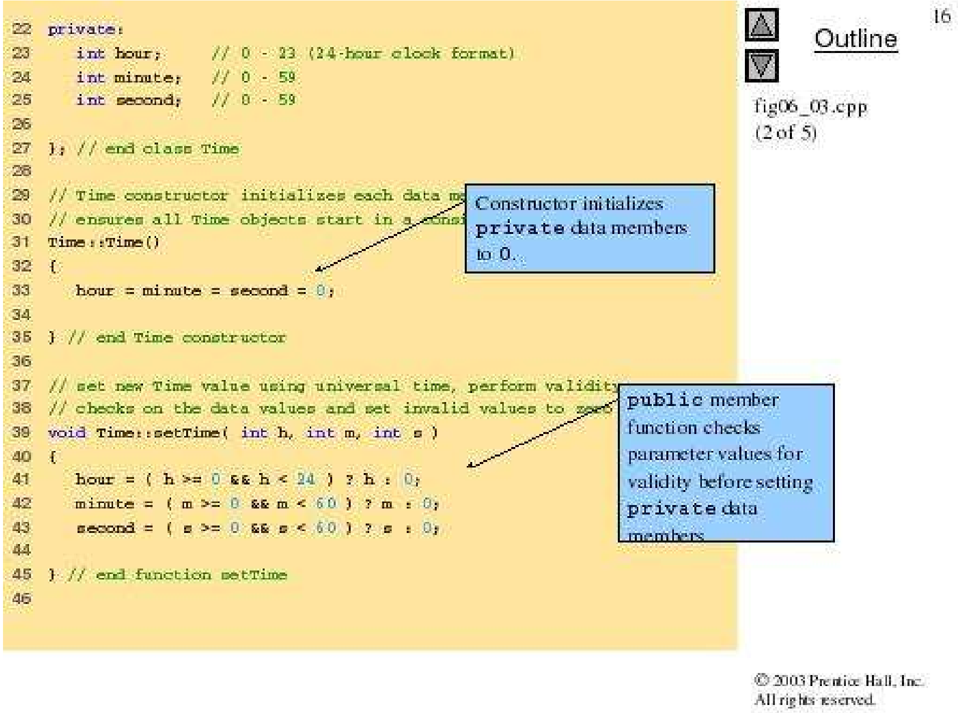
Outline

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**.

© 2003 Prentice Hall, Inc.
All rights reserved.



16

Outline

fig06_03.cpp
(2 of 5)

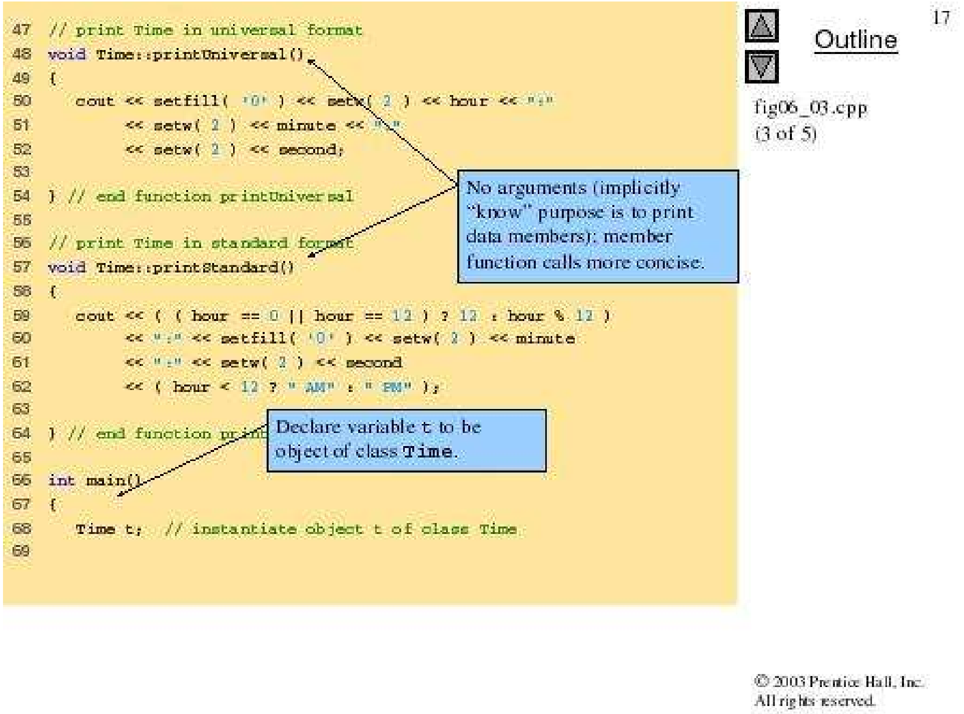
```
private:
   int hour;     // 0 - 23 (24-hour clock format)
   int minute;   // 0 - 59
   int second;   // 0 - 59

}; // 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

© 2003 Prentice Hall, Inc.
All rights reserved.

Figure 1.4: **Time** abstract data type implementation as a class, (part 1 of 3).



```
// 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

```

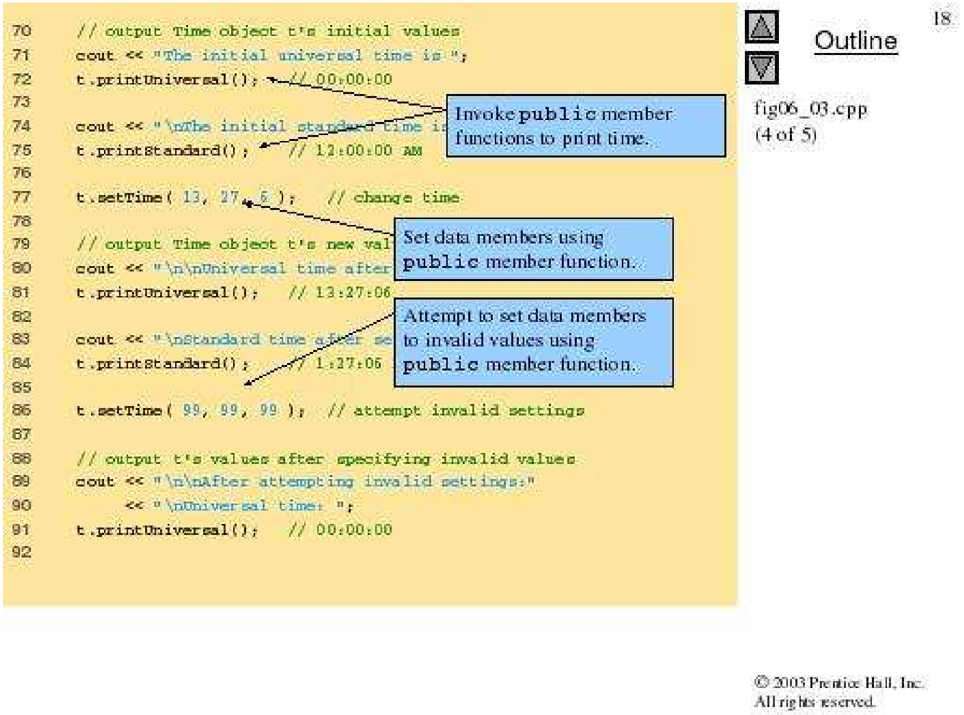
No arguments (implicitly "know" purpose is to print data members); member function calls more concise.

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

17

Outline

fig06_03.cpp
(3 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 i
   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.

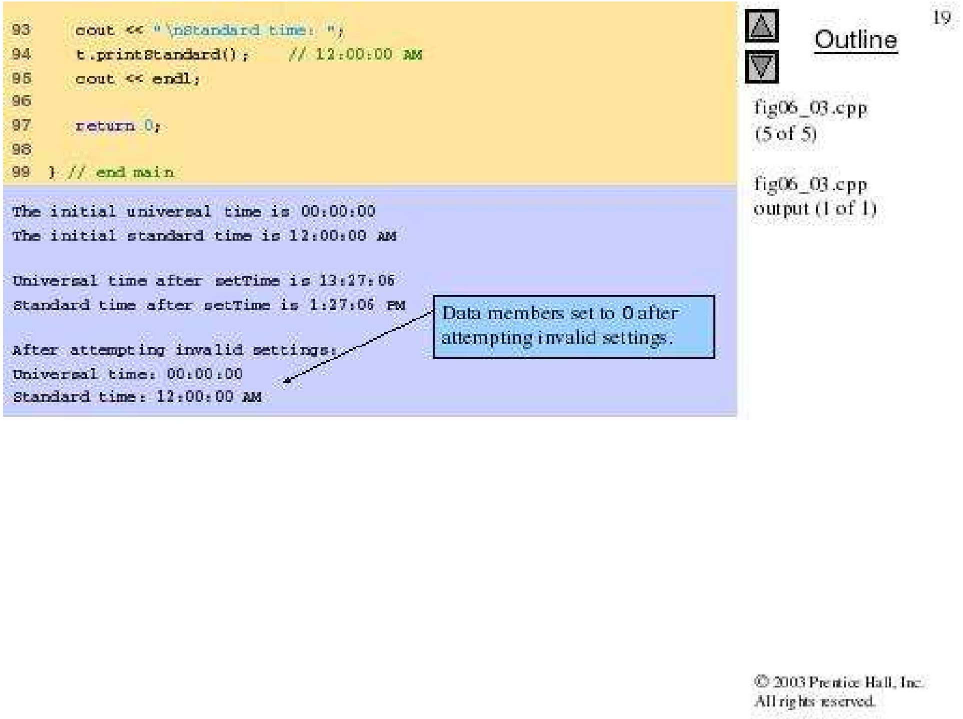
Attempt to set data members to invalid values using **public** member function.

18

Outline

fig06_03.cpp
(4 of 5)

Figure 1.5: **Time** abstract data type implementation as a class, (part 2 of 3).



19

```
   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)

© 2003 Prentice Hall, Inc.
All rights reserved.

Figure 1.6: **Time** abstract data type implementation as a class, (part 3 of 3).

  - 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 **struct**s
  - Dot member selection operator (**.**)
    - Object
    - Reference to object
  - Arrow member selection operator (− >)

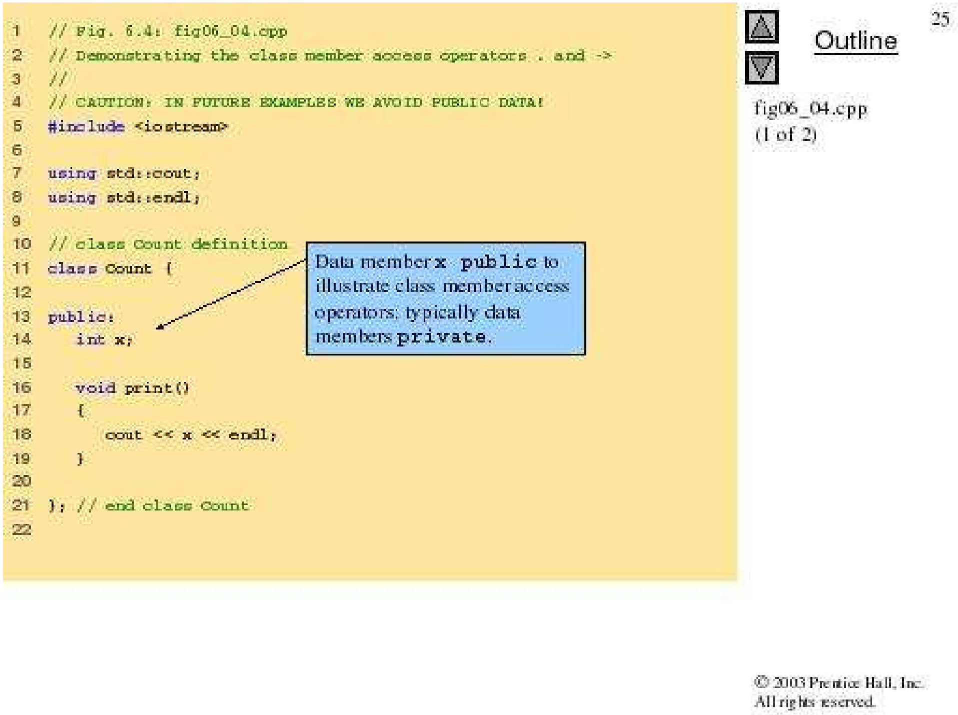
    * pointers

1.8 Separating Interface from Implementation (see Figs 1.8-1.11)

- 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

1.9 Controlling Access to Members (see Fig. 1.12)

- Access modes
  - **private**
    * Default access mode
    * Accessible to member functions and **friend**s
  - **public**



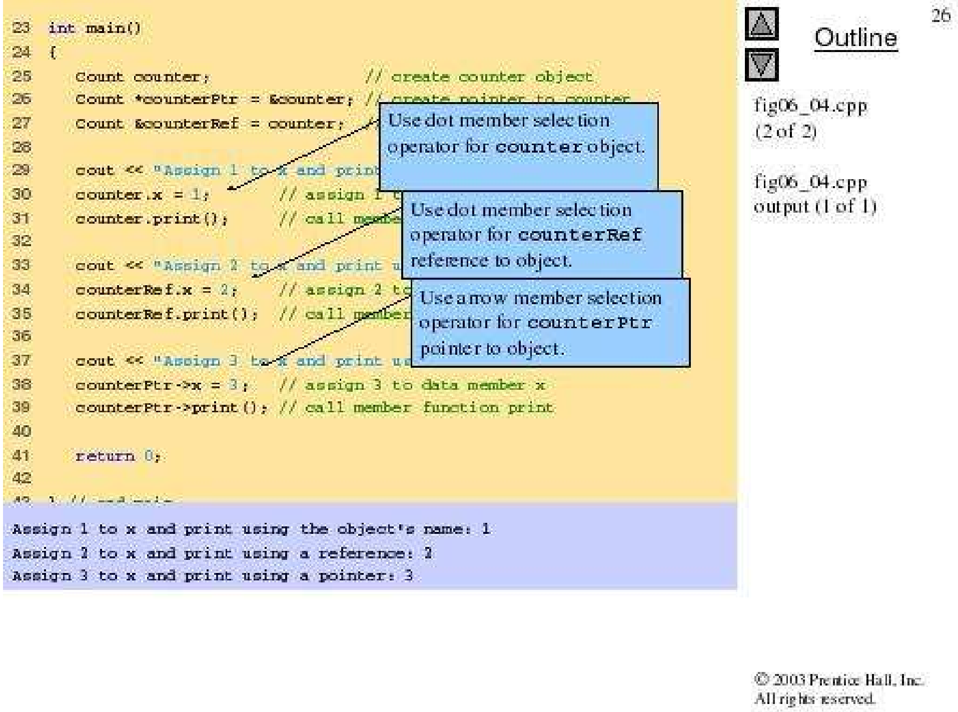
25

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**.



26

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 print
   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
```

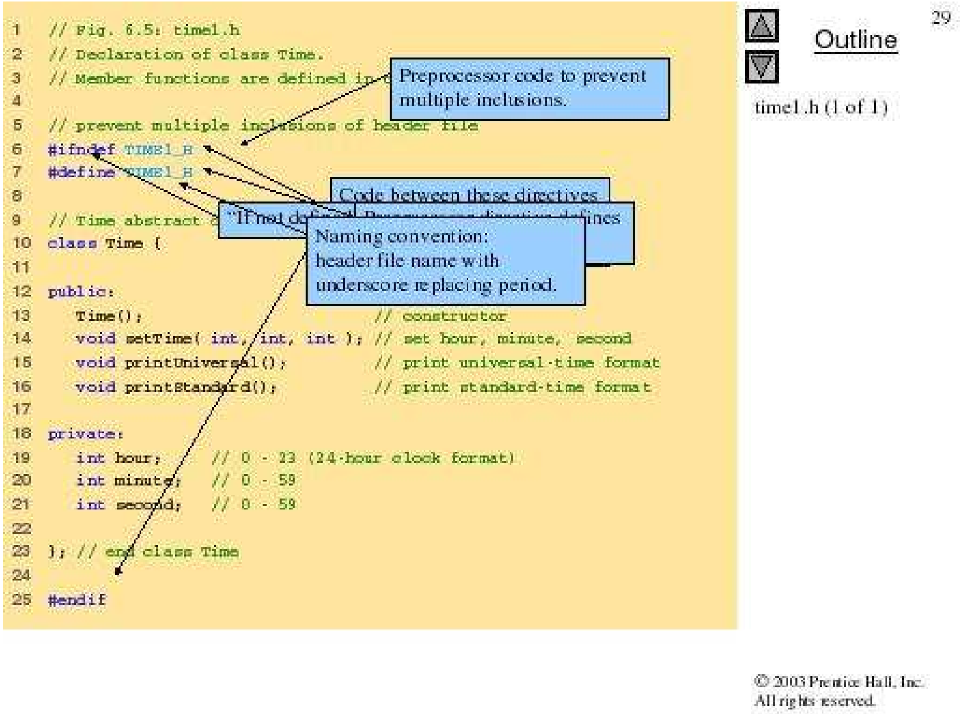
```
Assign 1 to x and print using the object's name: 1
Assign 2 to x and print using a reference: 2
Assign 3 to x and print using a pointer: 3
```

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.

Figure 1.7: Demonstrating the class member access operators. and − >



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

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

// Time abstract
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 defined"

Preprocessor directive defines

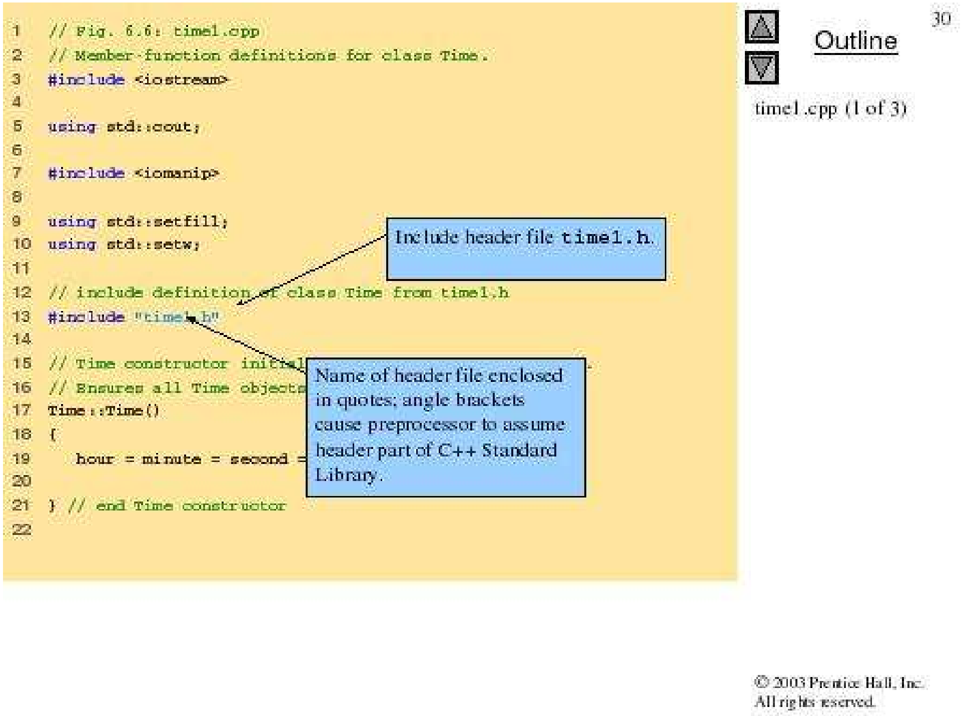
Naming convention: header file name with underscore replacing period.

© 2003 Prentice Hall, Inc. All rights reserved.

Figure 1.8: **Time** class definition

  - ∗ Accessible to any function in program with handle to class object
  - ∗ **protected** ; (discuss later)

- 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



```
// 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 initial
// Ensures all Time objects
Time::Time()
{
   hour = minute = second =

} // end Time constructor

```

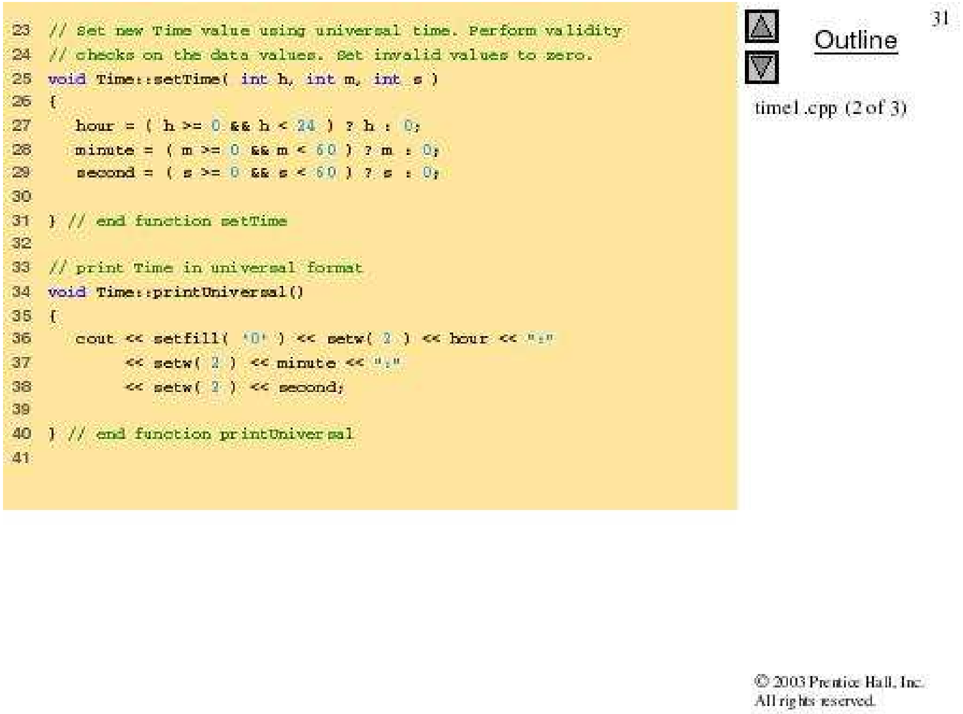
Include header file **time1.h**.

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

Outline

time1.cpp (1 of 3)

30



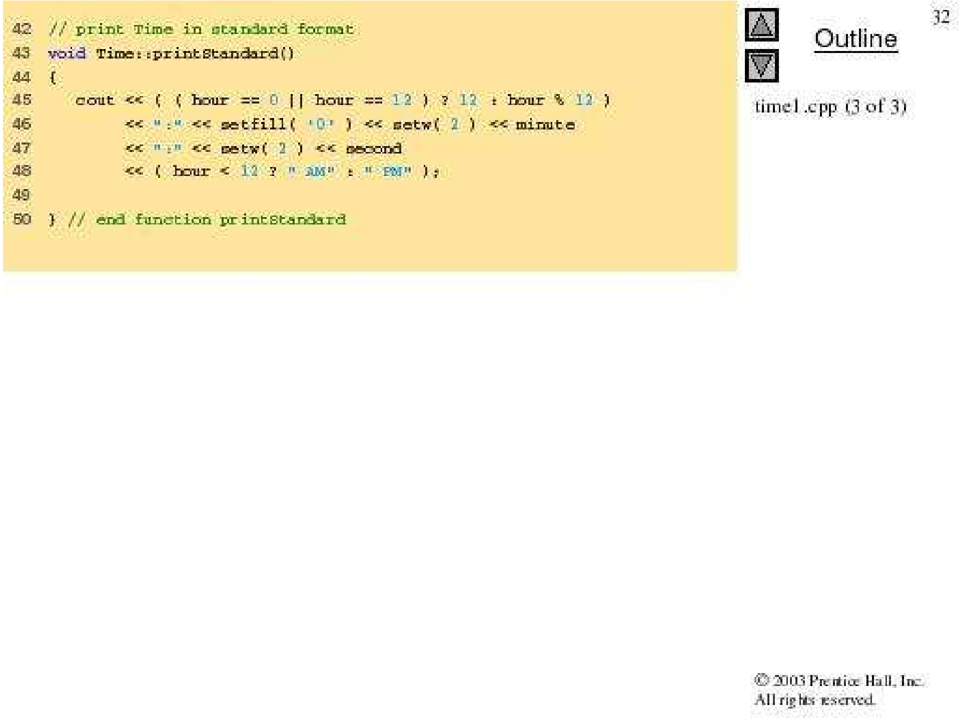
```
// Set new Time value using universal time. Perform validity
// checks on the data values. Set invalid values to zero.
void Time::setTime( int h, int m, int s )
{
   hour = ( h >= 0 && h < 24 ) ? h : 0;
   minute = ( m >= 0 && m < 60 ) ? m : 0;
   second = ( s >= 0 && s < 60 ) ? s : 0;

} // 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

```

Outline

time1.cpp (2 of 3)

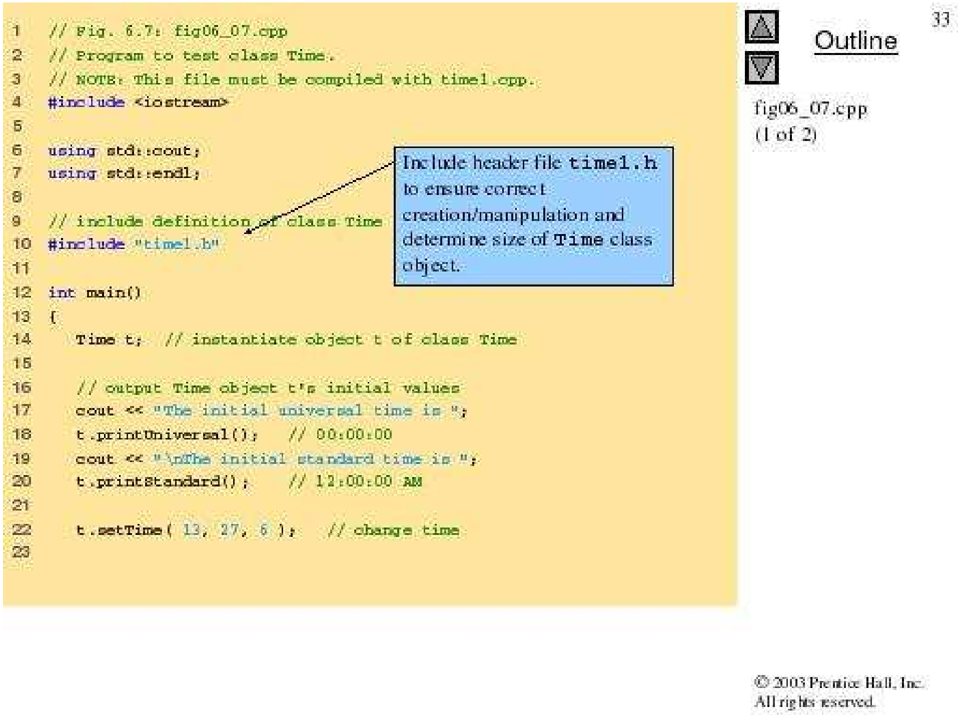
31

Figure 1.9: **Time** class member-function definitions (part 1 of 2).



```
// 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
```

Figure 1.10: **Time** class member-function definitions (part 2 of 2).



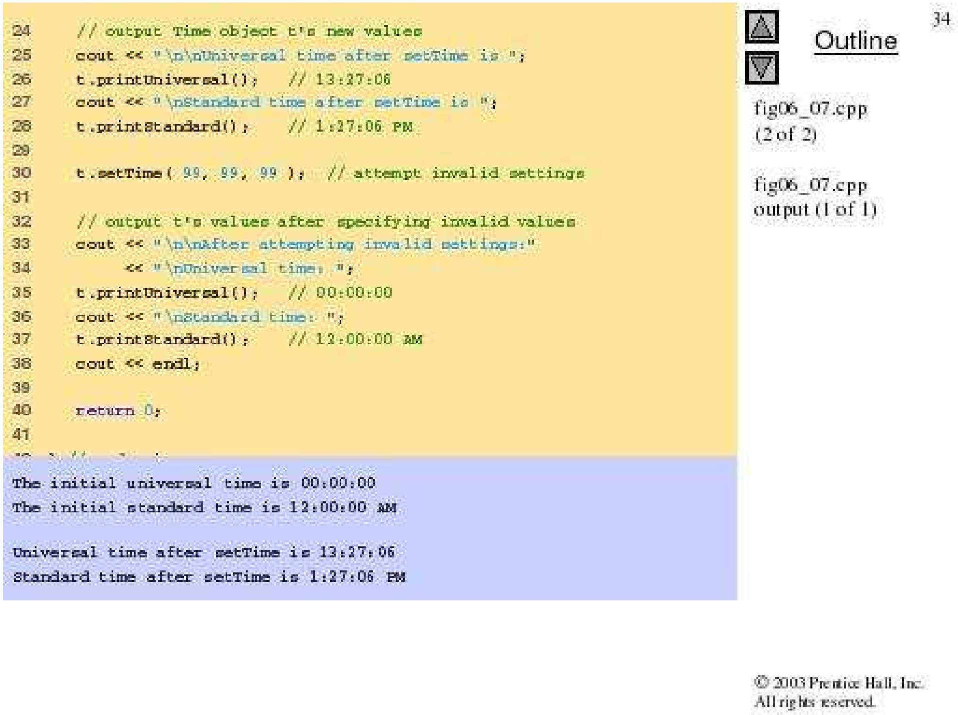
33

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.



34

Outline

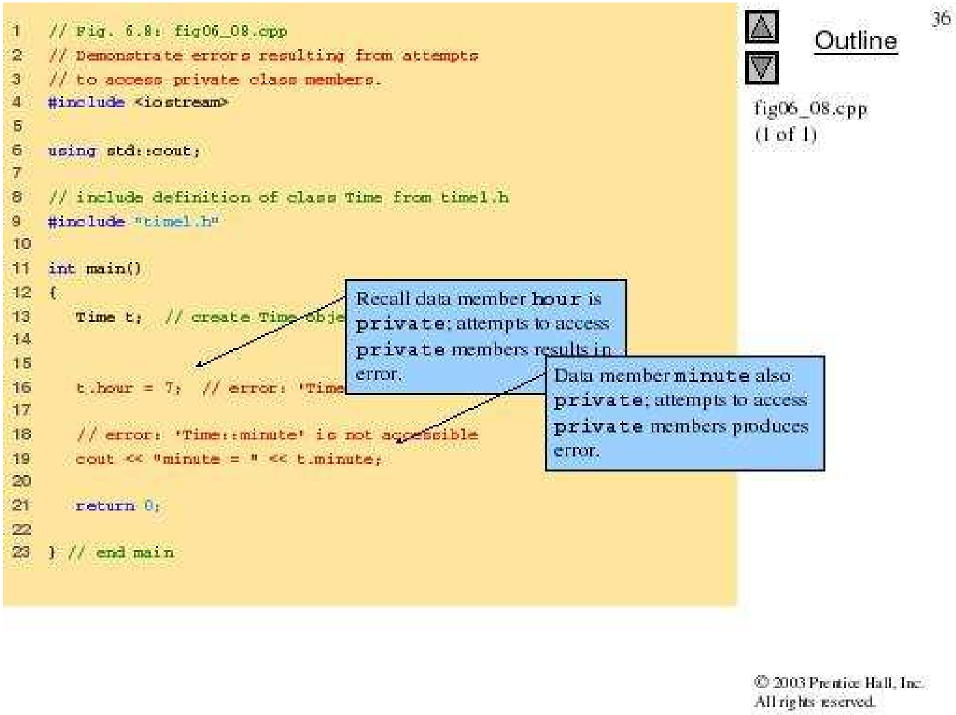
fig06_07.cpp
(2 of 2)

fig06_07.cpp
output (1 of 1)

```
   // 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;

```

```
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
```

Figure 1.11: Program to test class **Time**.



36

Outline

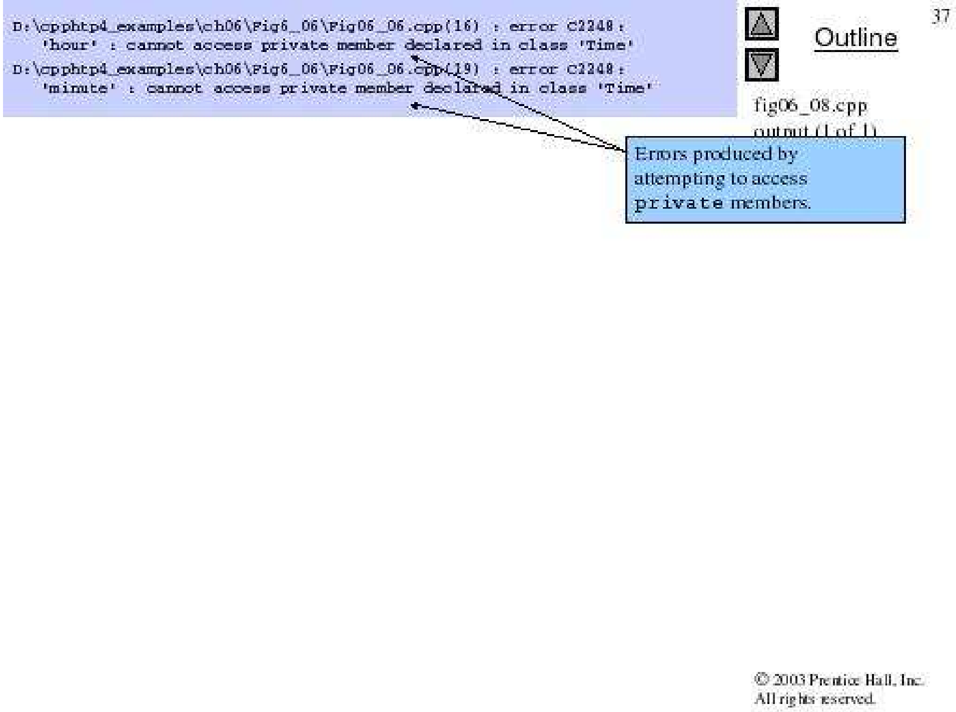
fig06_08.cpp
(1 of 1)

```
// 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.

© 2003 Prentice Hall, Inc.
All rights reserved.



37

Outline

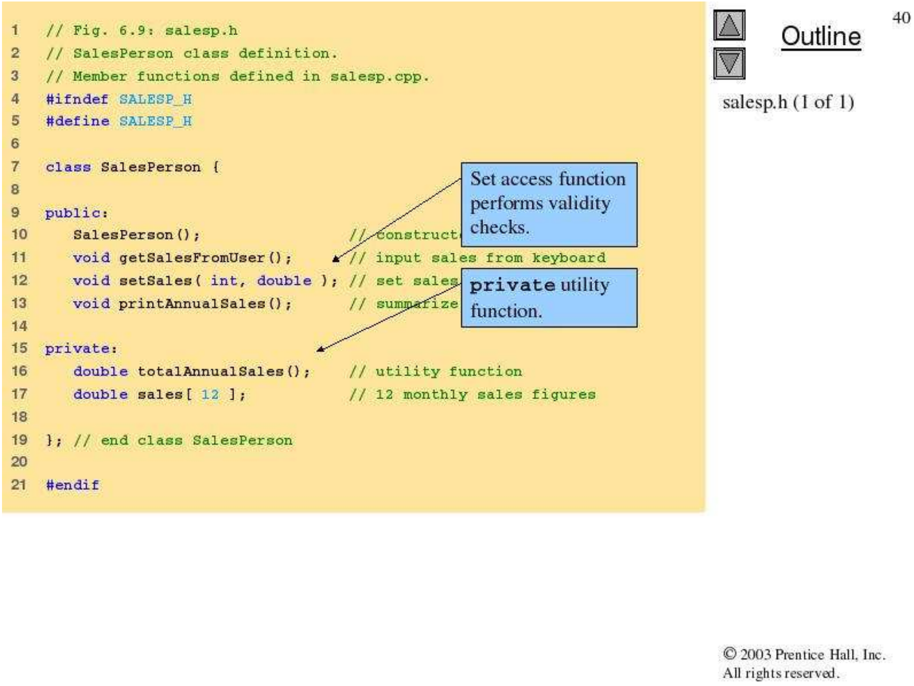
fig06_08.cpp
output (1 of 1)

```
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.

© 2003 Prentice Hall, Inc.
All rights reserved.

Figure 1.12: **private** members of a class are not accessible outside the class.



```
// 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.

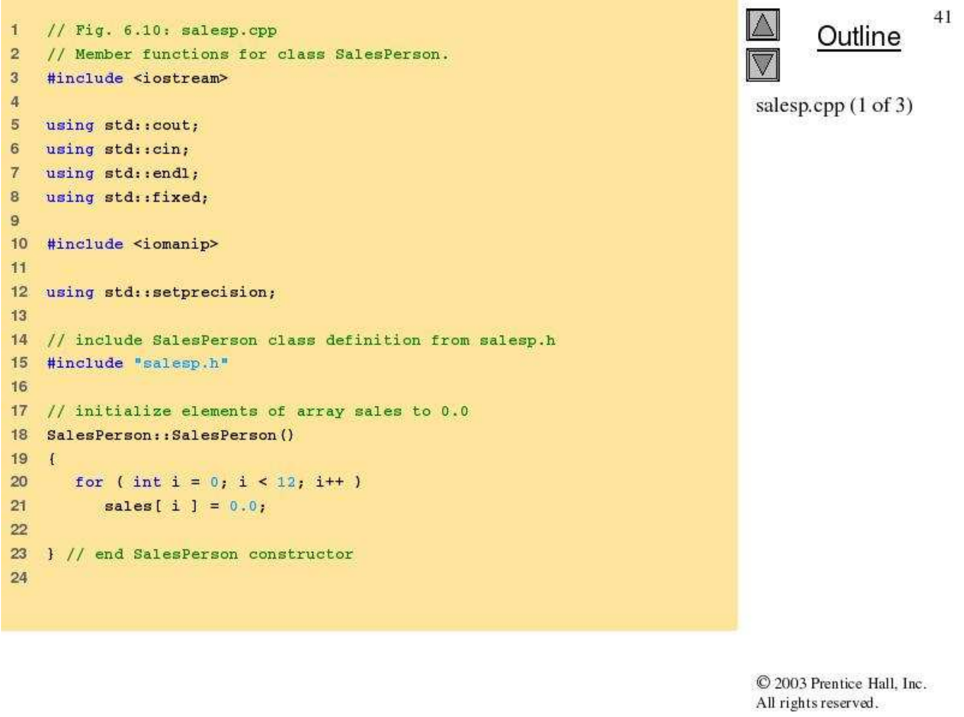
Figure 1.13: **SalesPerson** class definition

1.10 Access Functions and Utility Functions

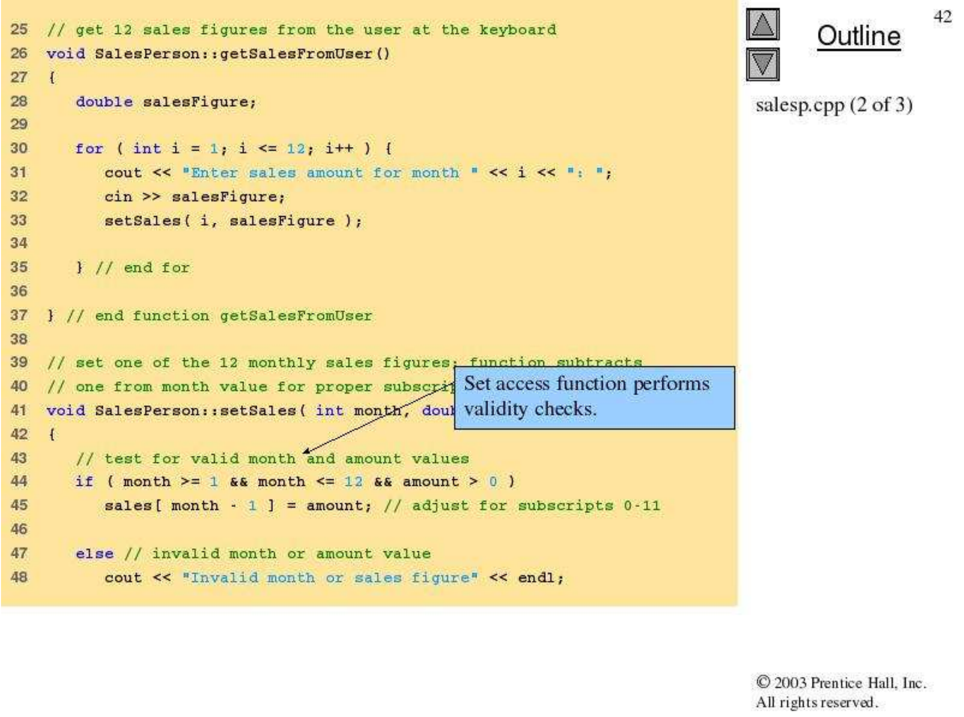
Not all member functions need be made **public** to serve as part of the interface of the class.

- 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

The program of Figs. 1.13-1.16 demonstarates the notion of a *utility function* (also called helper function).



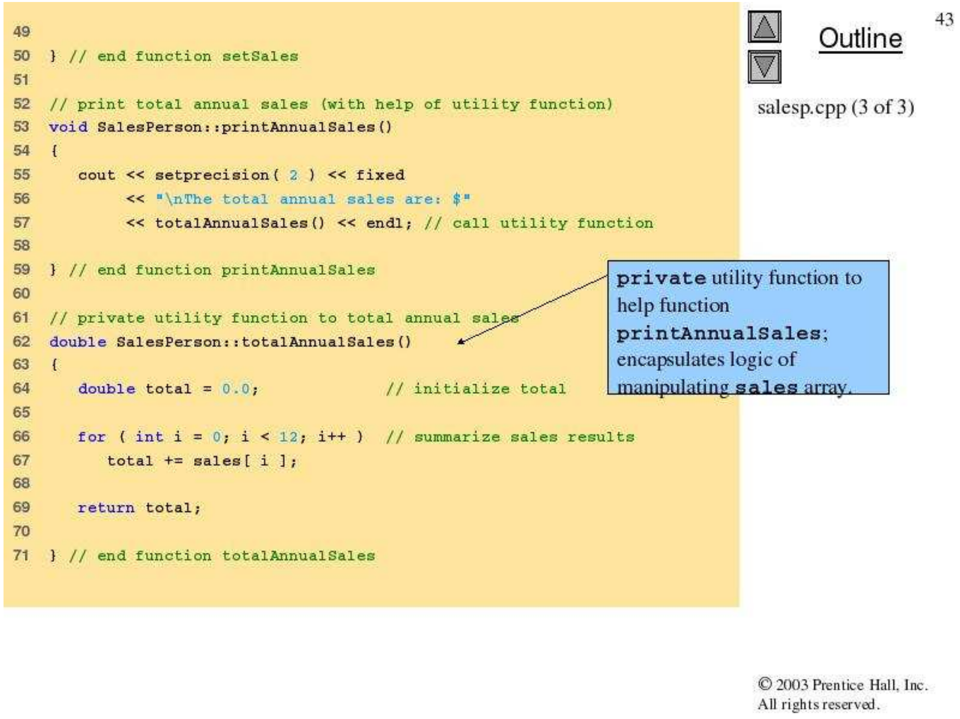
```cpp
// Fig. 6.10: salesp.cpp
// Member functions for class SalesPerson.
#include <iostream>

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

```



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

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

   } // end for

} // end function getSalesFromUser

// set one of the 12 monthly sales figures; function subtracts
// one from month value for proper 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.

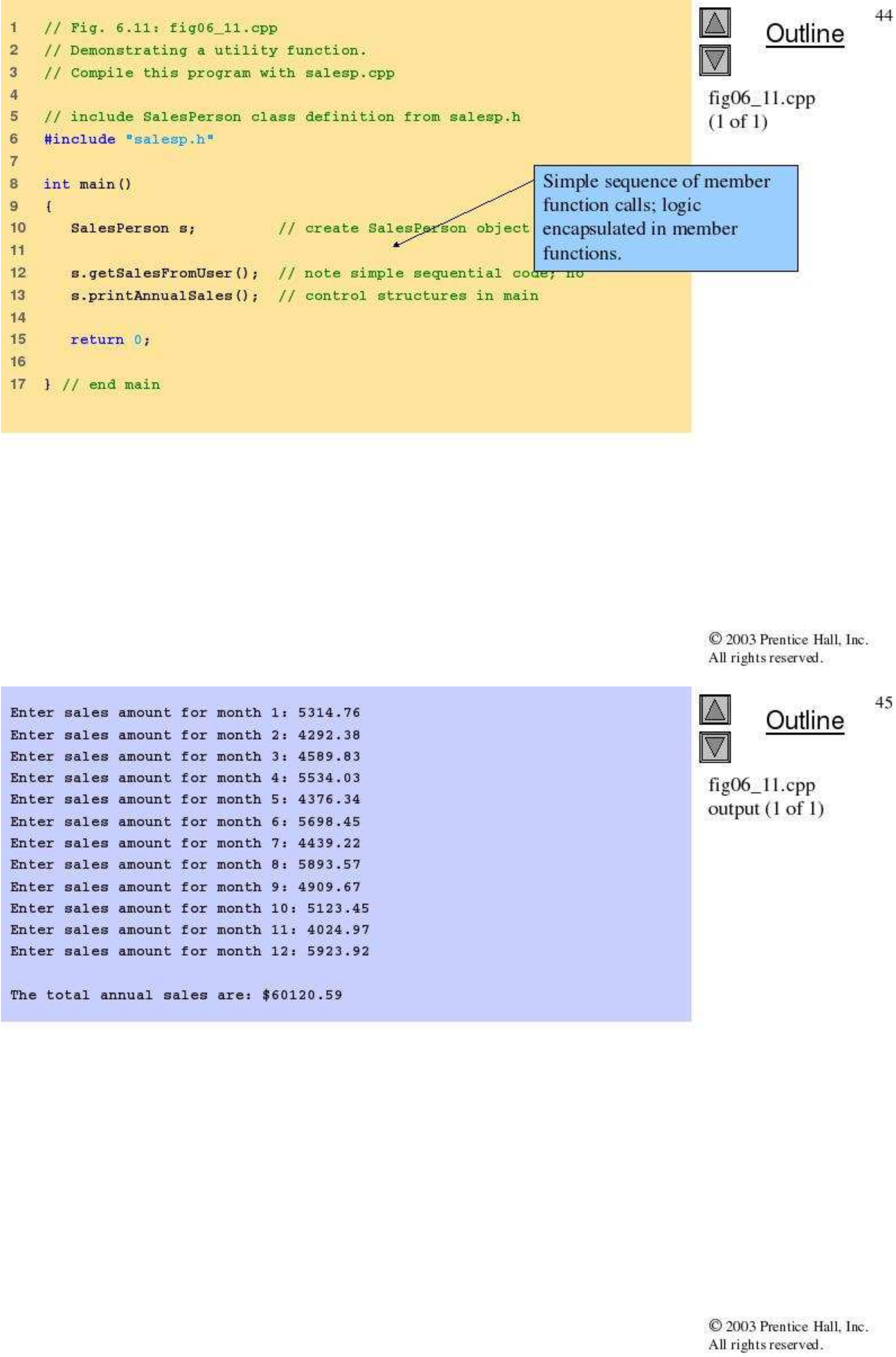
Figure 1.14: **SalesPerson** class member-function definitions (part 1 of 2)



```

} // 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.

Figure 1.15: **SalesPerson** class member-function definitions (part 2 of 2)



```
// 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 code; no
   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
(1 of 1)

44

© 2003 Prentice Hall, Inc.
All rights reserved.

```
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

© 2003 Prentice Hall, Inc.
All rights reserved.

Figure 1.16: Utility function demonstration

1.11 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

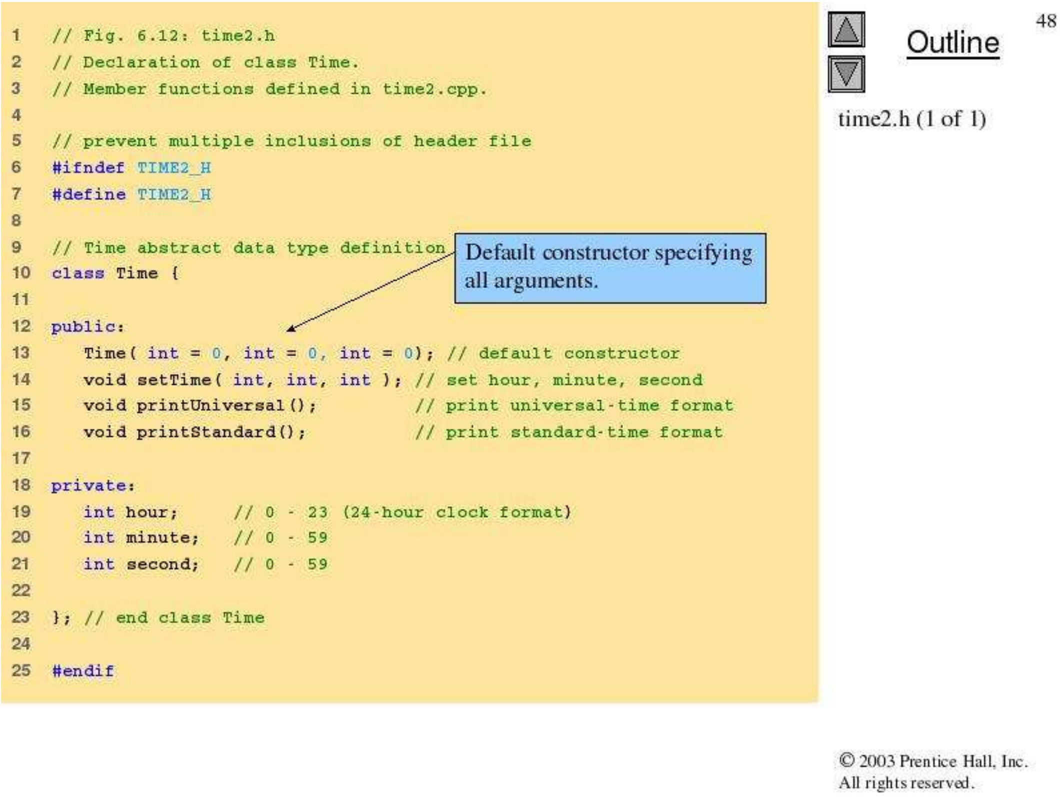
    ```
    Class-type ObjectName( value1,value2,...};
    ```

  The programmer provides the constructor, which is then invoked each time an object of that class is created (instantiated).

1.12 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

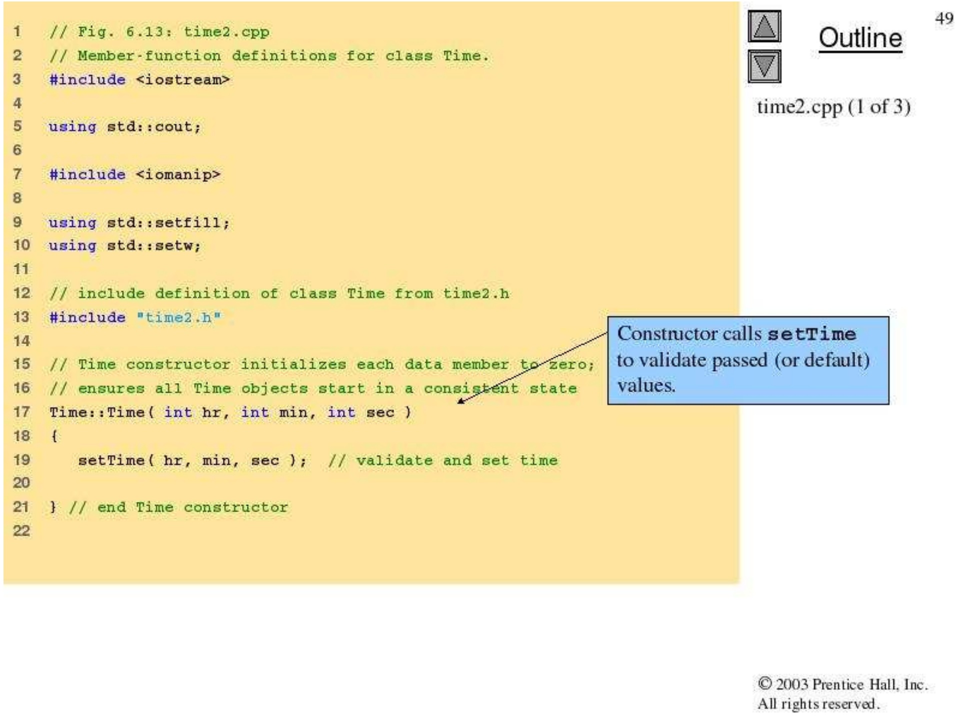
The program of Figs. 1.17-1.21 enhances class **Time** to demonstrate how arguments are implicitly passed to a constructor.



```
// 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.

Figure 1.17: **Time** class containing a constructor with default arguments.



49

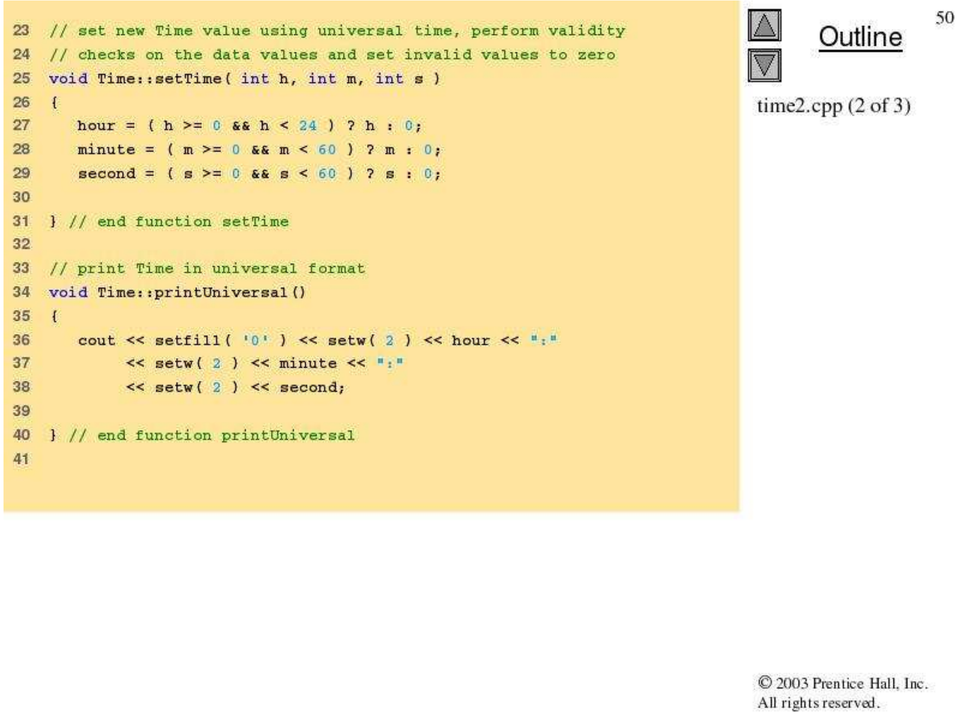
Outline

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.

© 2003 Prentice Hall, Inc.
All rights reserved.



50

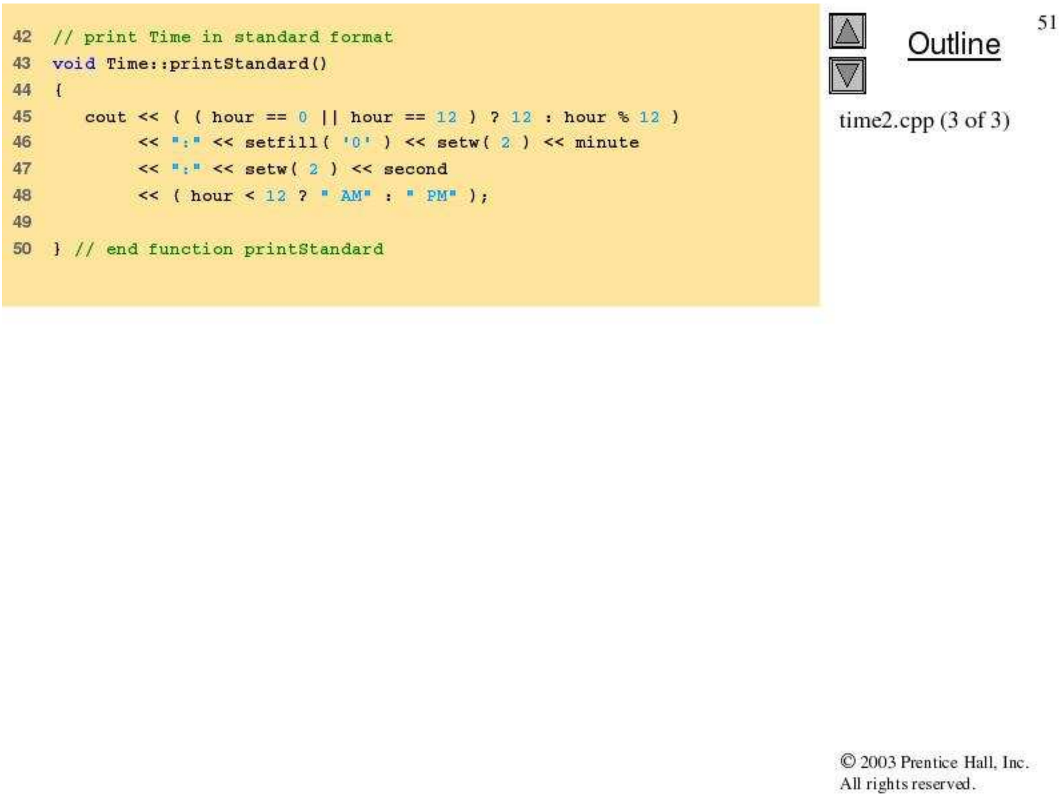
Outline

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

```

© 2003 Prentice Hall, Inc.
All rights reserved.

Figure 1.18: **Time** class member-function definitions including a constructor that takes arguments. (part 1 of 2)



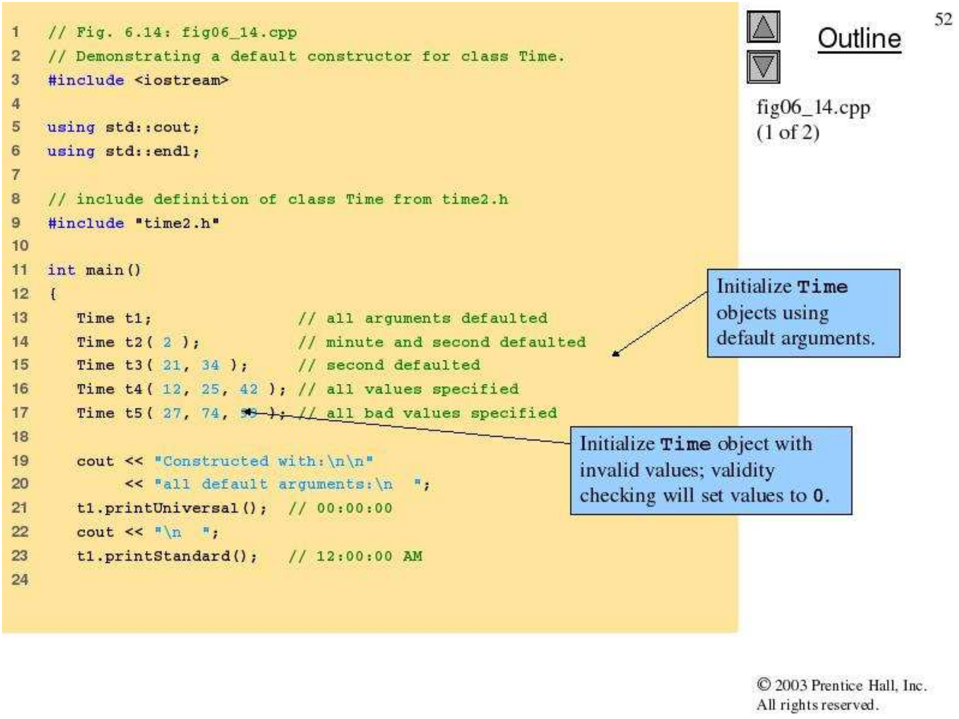
51

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
```

Figure 1.19: **Time** class member-function definitions including a constructor that takes arguments. (part 2 of 2)



```
// 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.

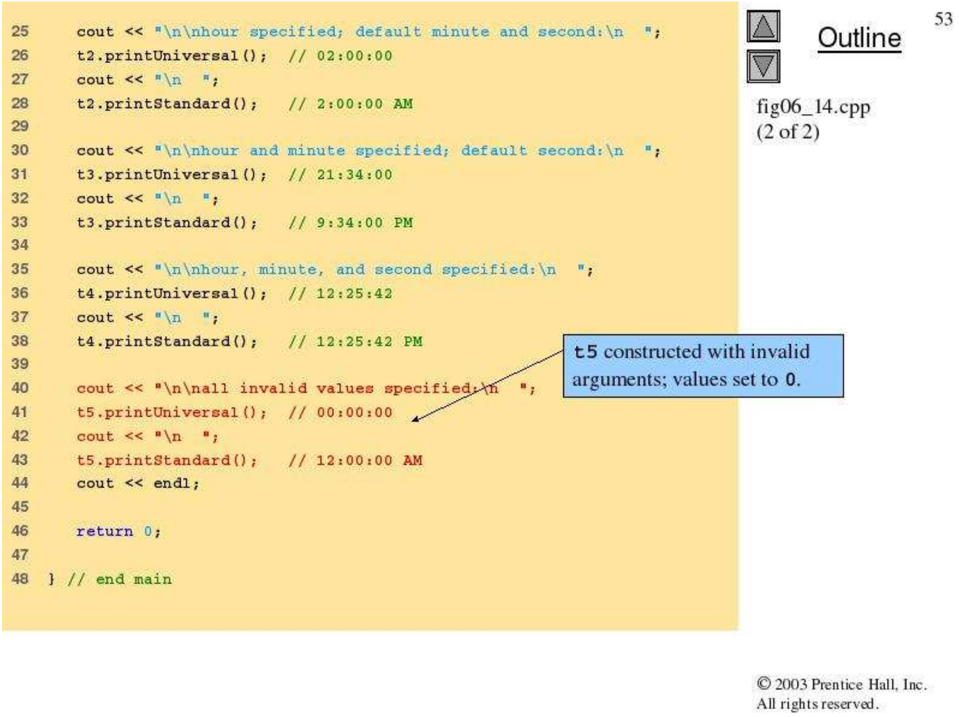
Initialize **Time** object with invalid values; validity checking will set values to **0**.

Outline

fig06_14.cpp
(1 of 2)

52

© 2003 Prentice Hall, Inc.
All rights reserved.



```
   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**.

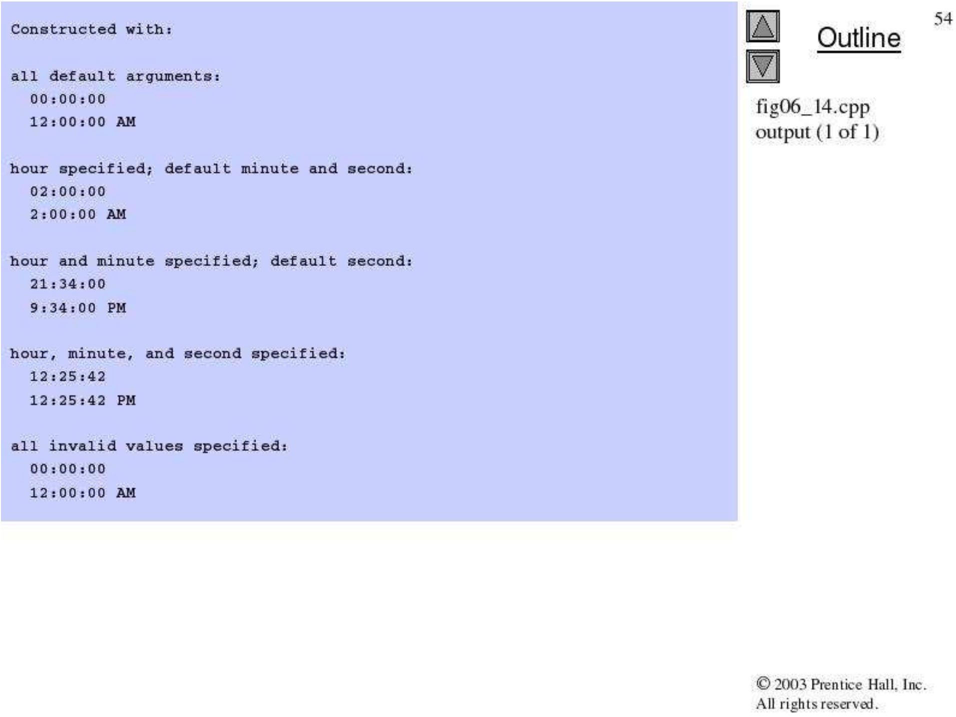
Outline

fig06_14.cpp
(2 of 2)

53

© 2003 Prentice Hall, Inc.
All rights reserved.

Figure 1.20: Constructor with default arguments. (part 1 of 2)



```
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
```

Figure 1.21: Constructor with default arguments. (part 2 of 2)

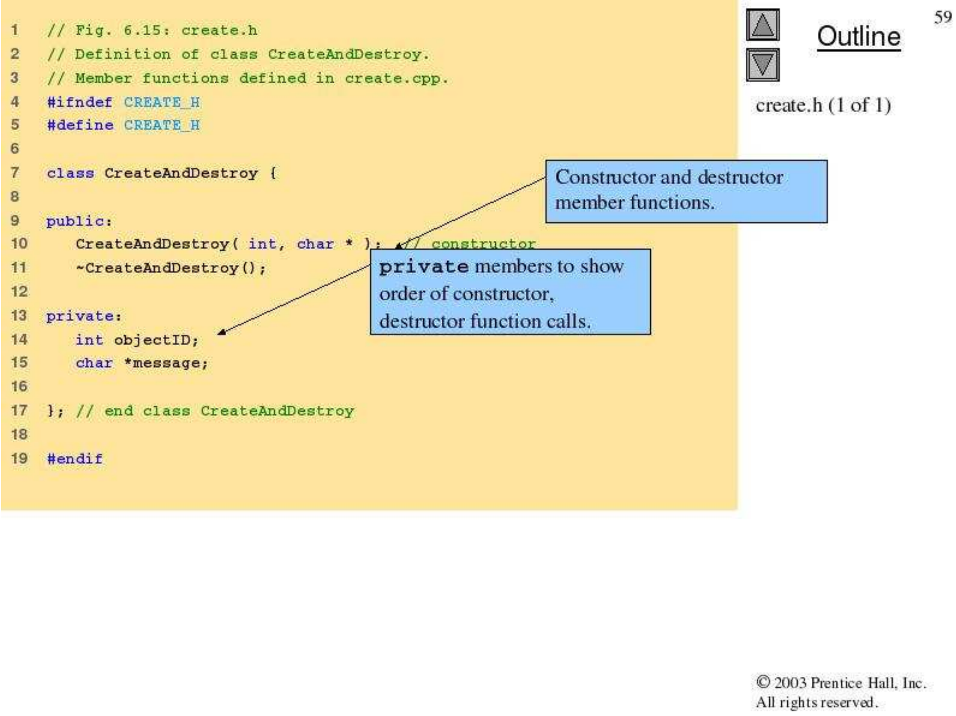
1.13 Destructors

- 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”

1.14 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
- 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**
    - **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**

The program of Figs. 1.22-1.25 demonstrates the order in which constructors and destructors are called for objects of class CreateAndDestroy of various storage classes in several scopes.



59

Outline

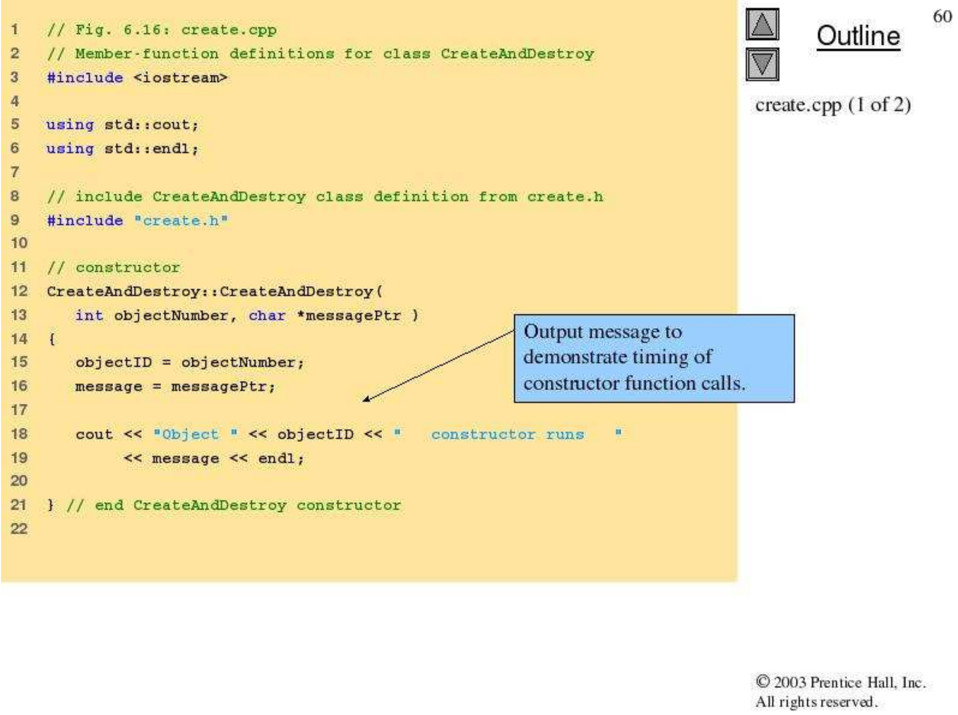
create.h (1 of 1)

```
// 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.

Figure 1.22: **CreateAndDestroy** class definition.



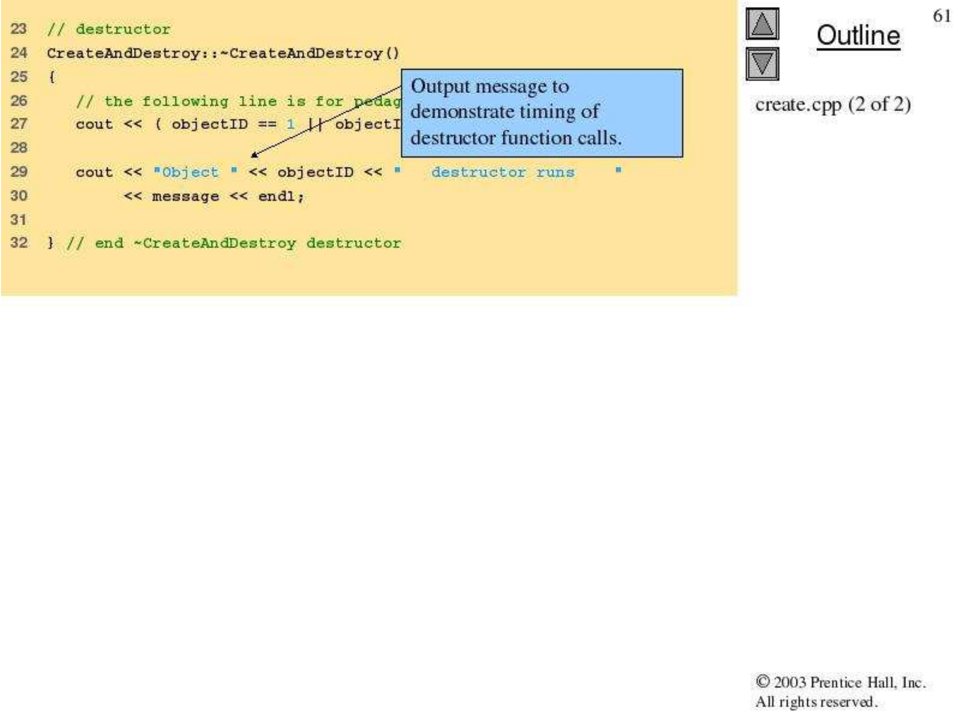
60

Outline

create.cpp (1 of 2)

```cpp
// 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.



61

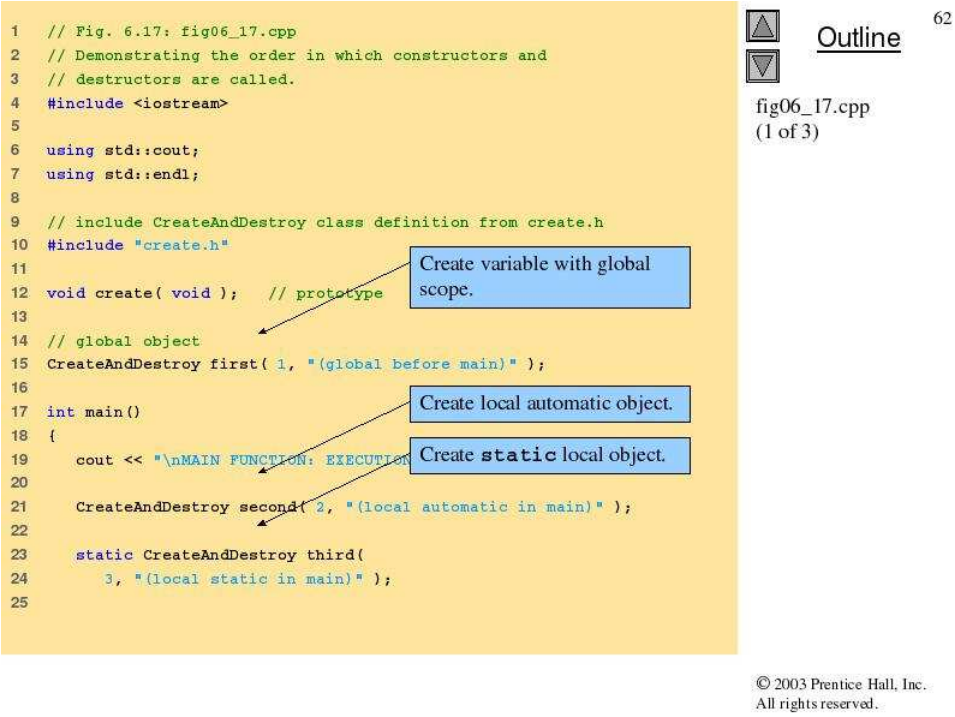
Outline

create.cpp (2 of 2)

```cpp
// 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.

Figure 1.23: **CreateAndDestroy** class member-function definitions.



```
// 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.

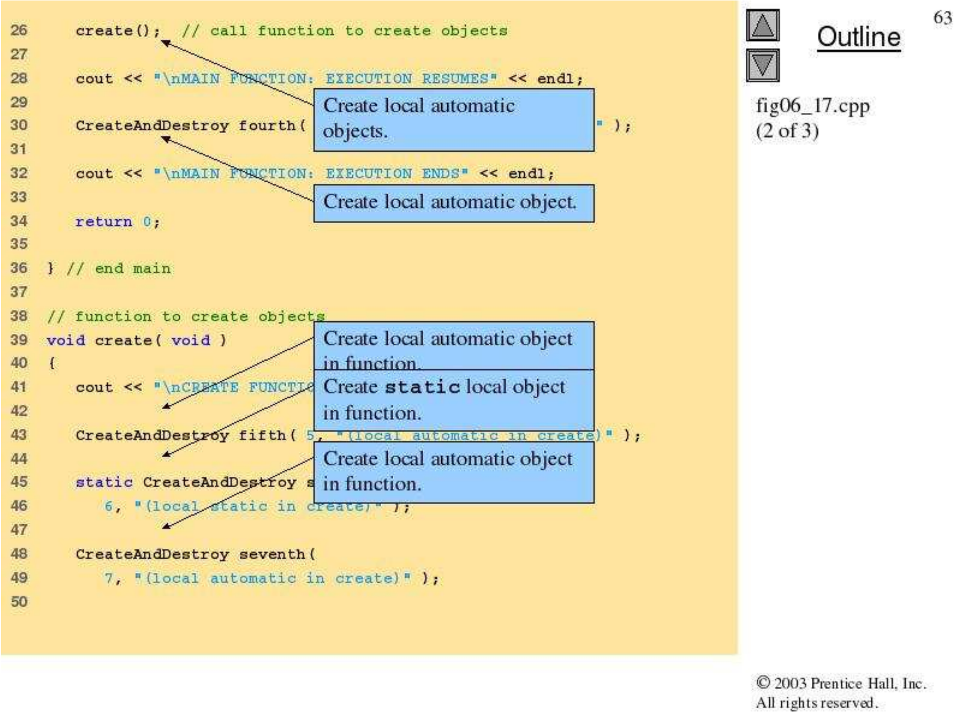
Create local automatic object.

Create **static** local object.

Outline

fig06_17.cpp
(1 of 3)

62



```
   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, "(local automatic in create)" );

   static CreateAndDestroy s
      6, "(local static in create)" );

   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.

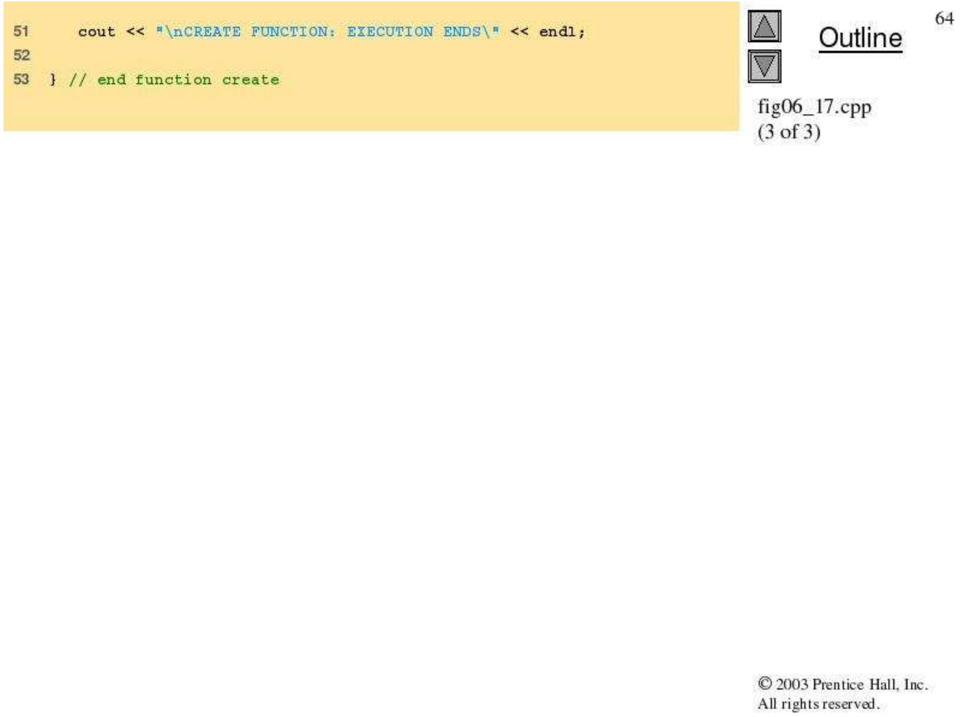
Create local automatic object in function.

Outline

fig06_17.cpp
(2 of 3)

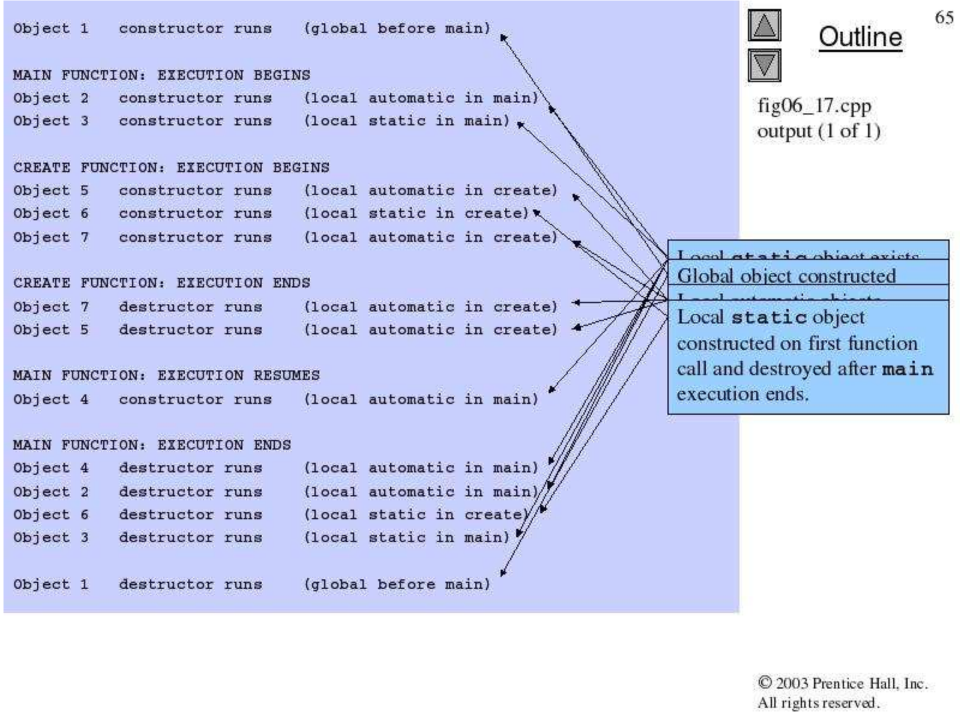
63

Figure 1.24: Order in which constructors and destructors are called. (part 1 of 2)



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

} // end function create
```

fig06_17.cpp
(3 of 3)



```
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)
```

fig06_17.cpp
output (1 of 1)

Global object constructed

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

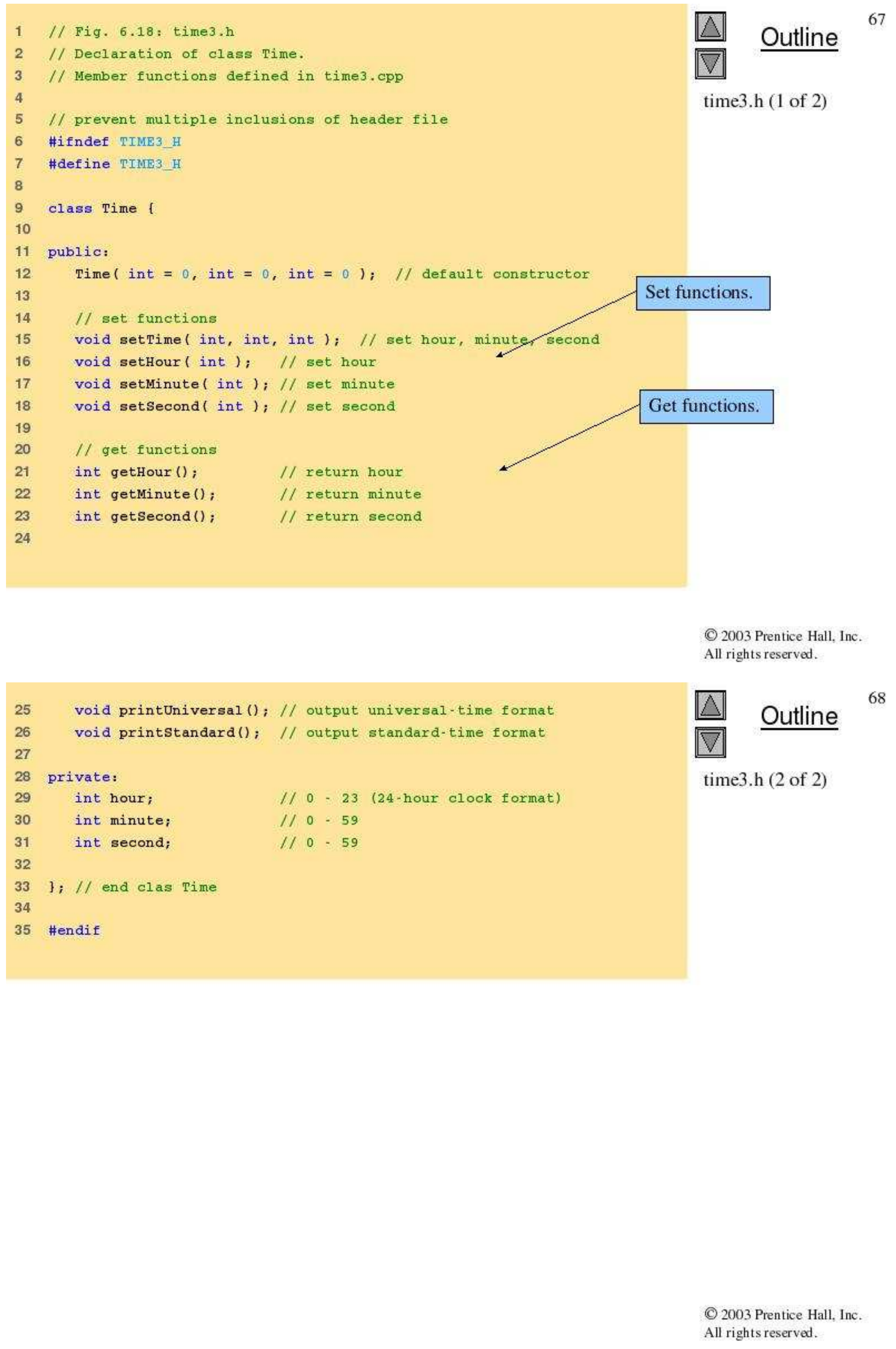
Figure 1.25: Order in which constructors and destructors are called. (part 2 of 2)

1.15 Using *Set* and *Get* Functions

A class's **private** data members can be accessed only by member functions (and friends) of the class. Classes often provide **public** member functions to allow clients of the class to *set* (i.e., write) or *get* (,.e., read) the values of **private** data members. These functions need not be called *set* and *get* specifically, but they often are.

- 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

The program of Figs. 1.26-1.30 enhances class **Time** to include *set* and *get* functions for the **private** data members **hour, minute,** and **second.**



67

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.

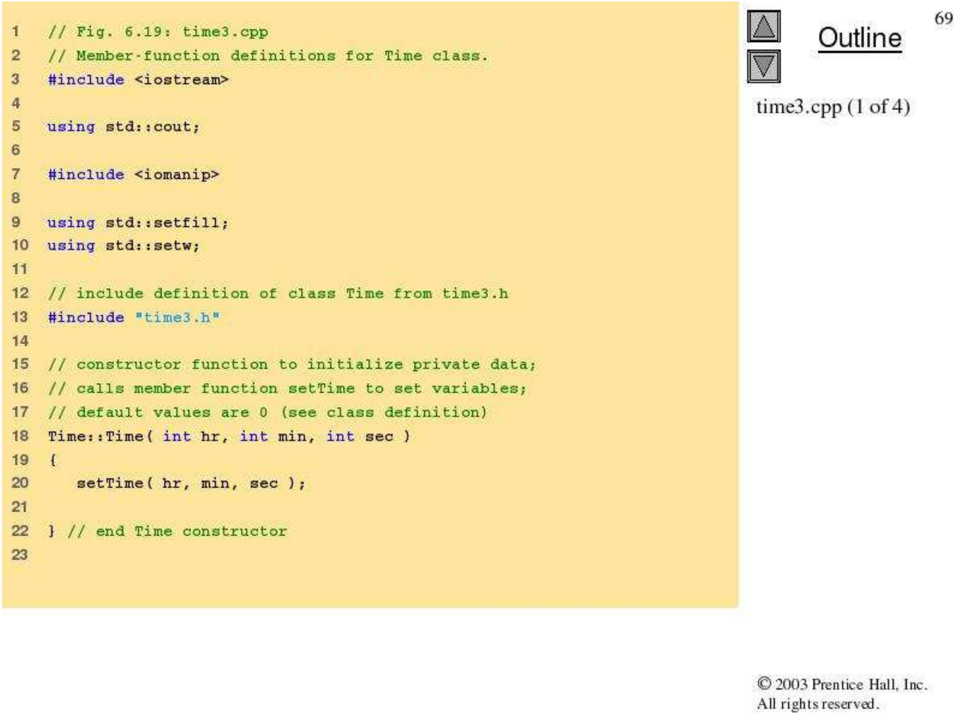
68

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
```

Figure 1.26: **Time** class definition with *set* and *get* functions.

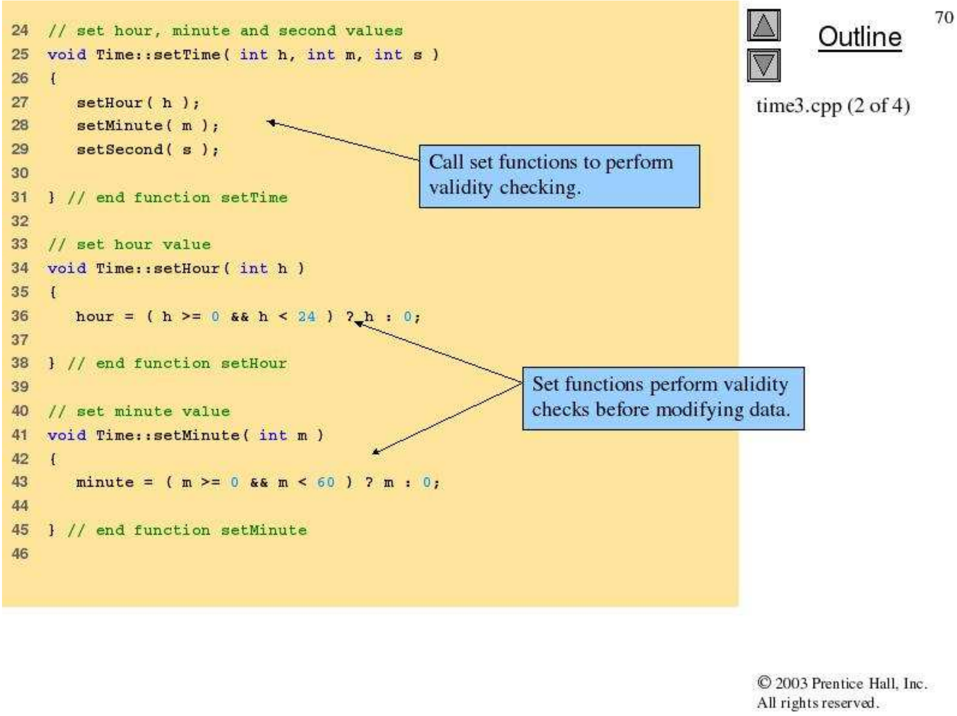


69

Outline

time3.cpp (1 of 4)

```
// 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

```



70

Outline

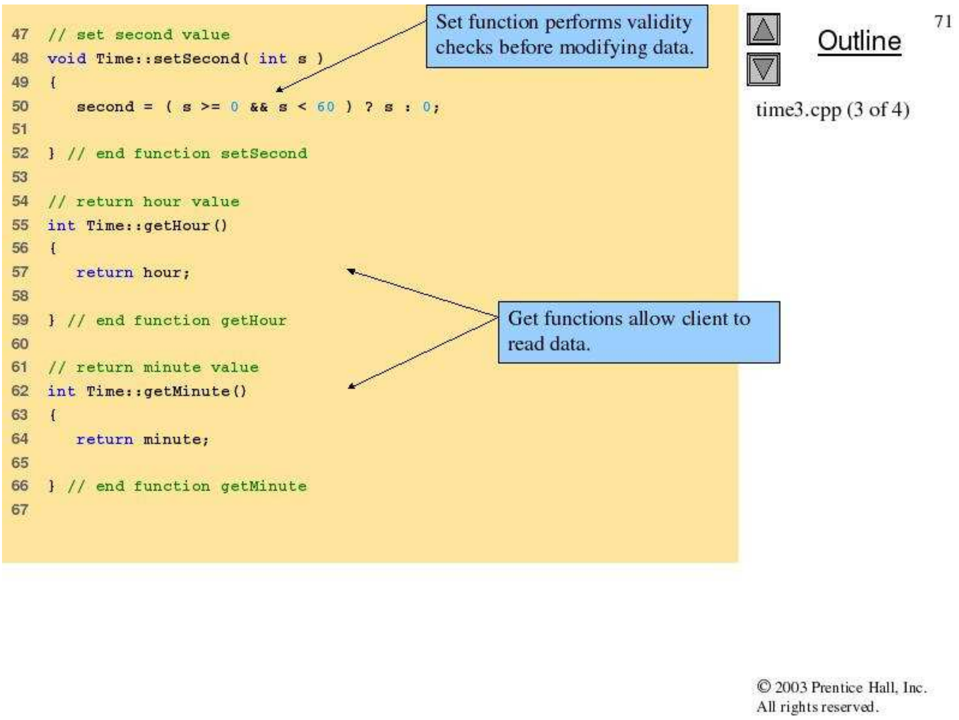
time3.cpp (2 of 4)

```
// 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.

Figure 1.27: **Time** class member-function definitions,including *set* and *get* functions. (part 1 of 2)



71

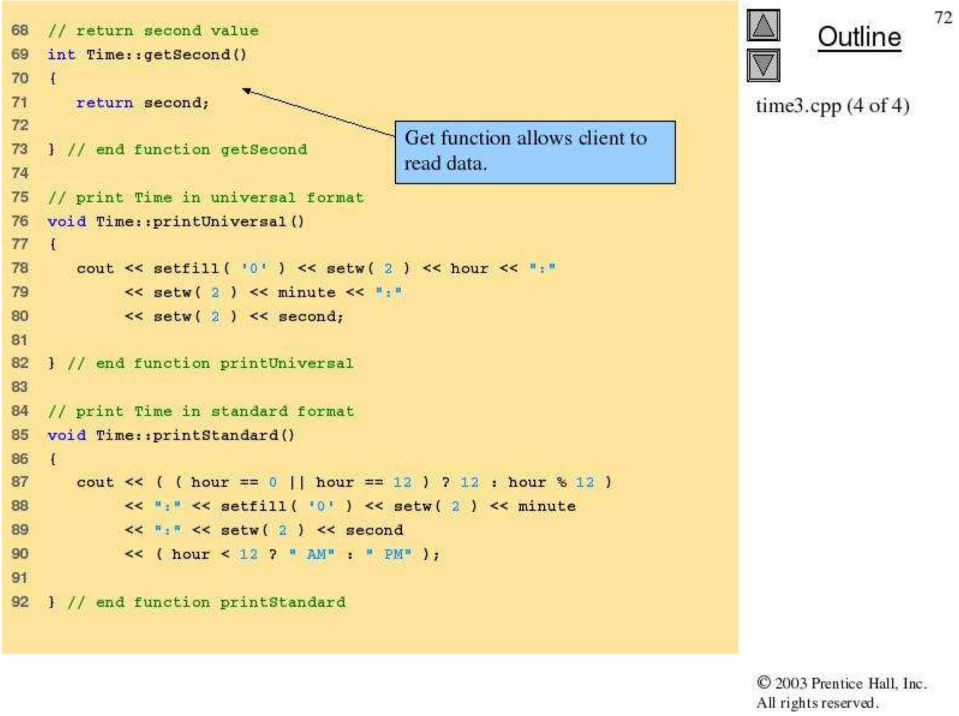
Outline

time3.cpp (3 of 4)

```cpp
// 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.



72

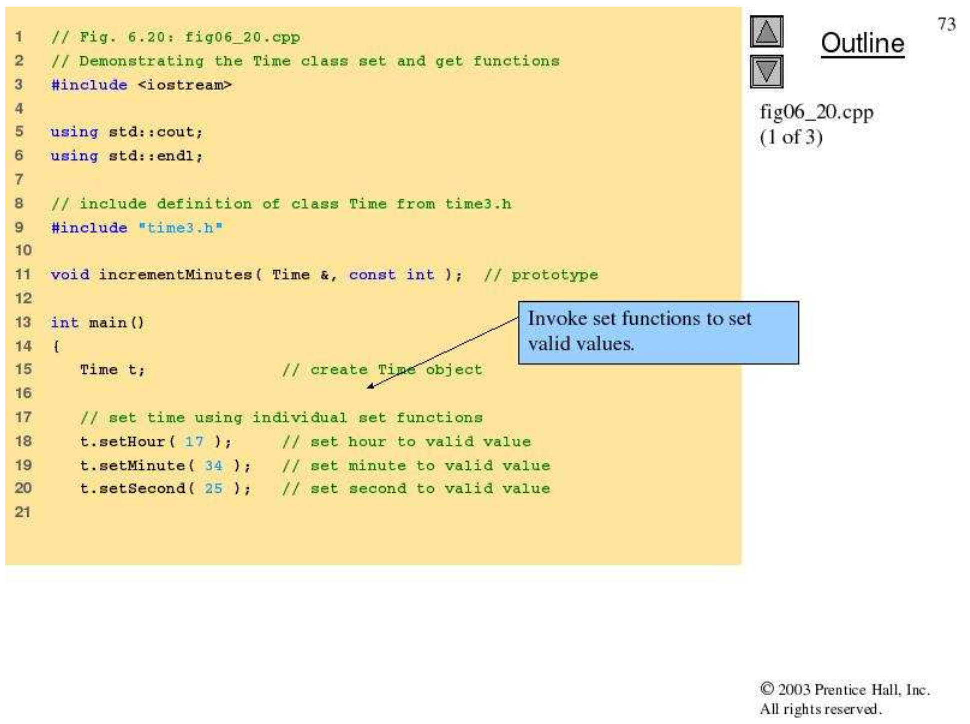
Outline

time3.cpp (4 of 4)

```cpp
// 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.

Figure 1.28: **Time** class member-function definitions,including *set* and *get* functions. (part 2 of 2)



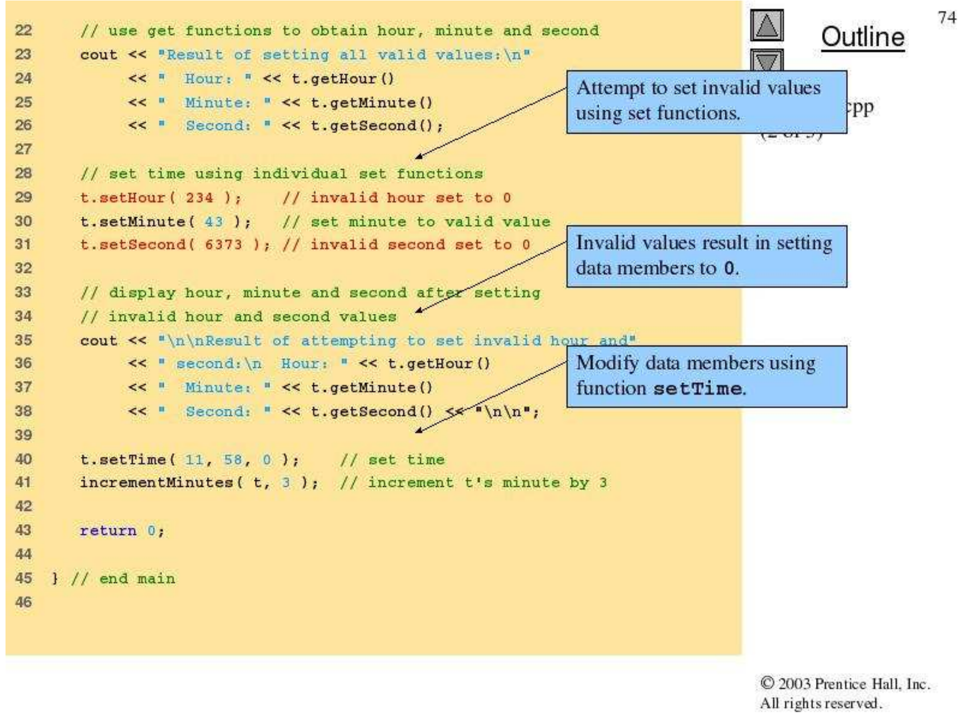
73

Outline

fig06_20.cpp
(1 of 3)

```cpp
// 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.



74

Outline

pp

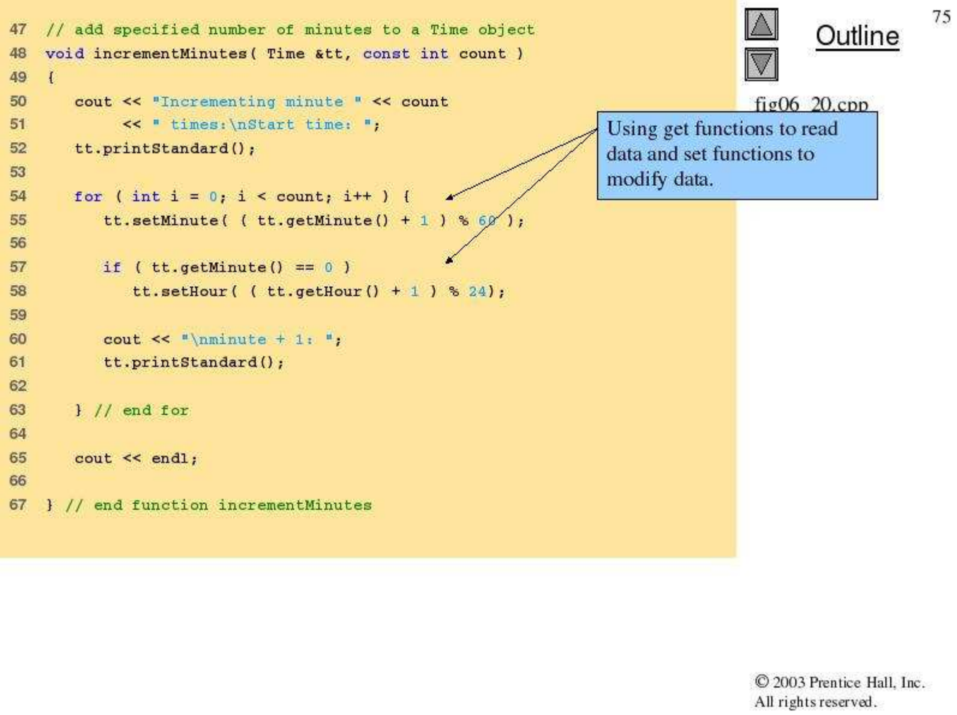
```cpp
   // use get functions to obtain hour, minute and second
   cout << "Result of setting all valid values:\n"
        << "  Hour: " << t.getHour()
        << "  Minute: " << t.getMinute()
        << "  Second: " << t.getSecond();

   // 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**.

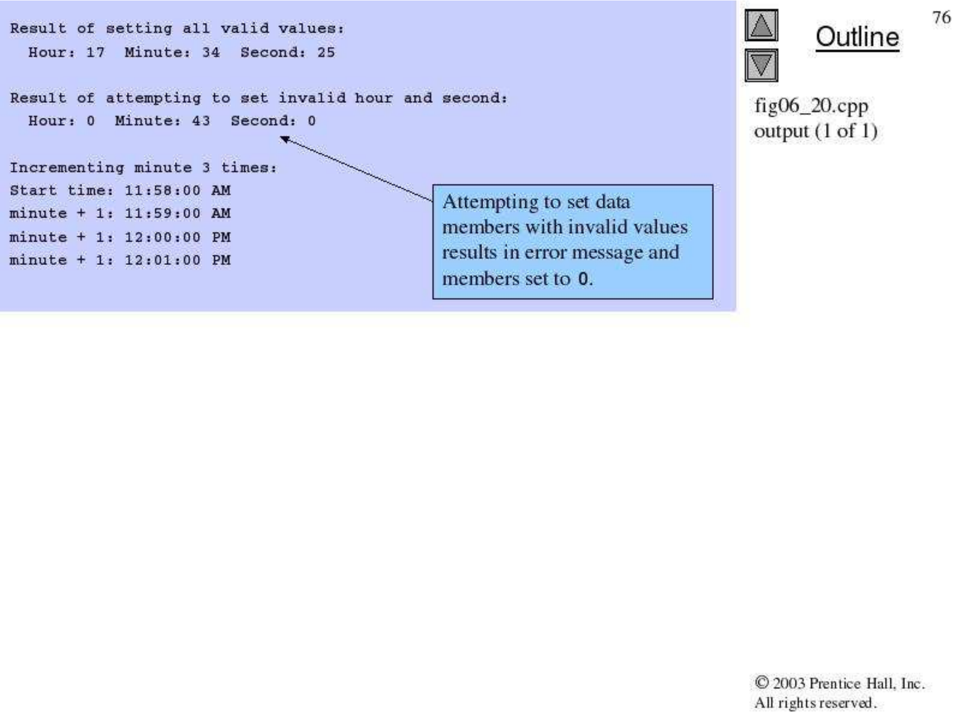
Modify data members using function **setTime**.

Figure 1.29: *Set* and *get* functions manipulating an object's **private** data. (part 1 of 2)



```
// 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.



```
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**.

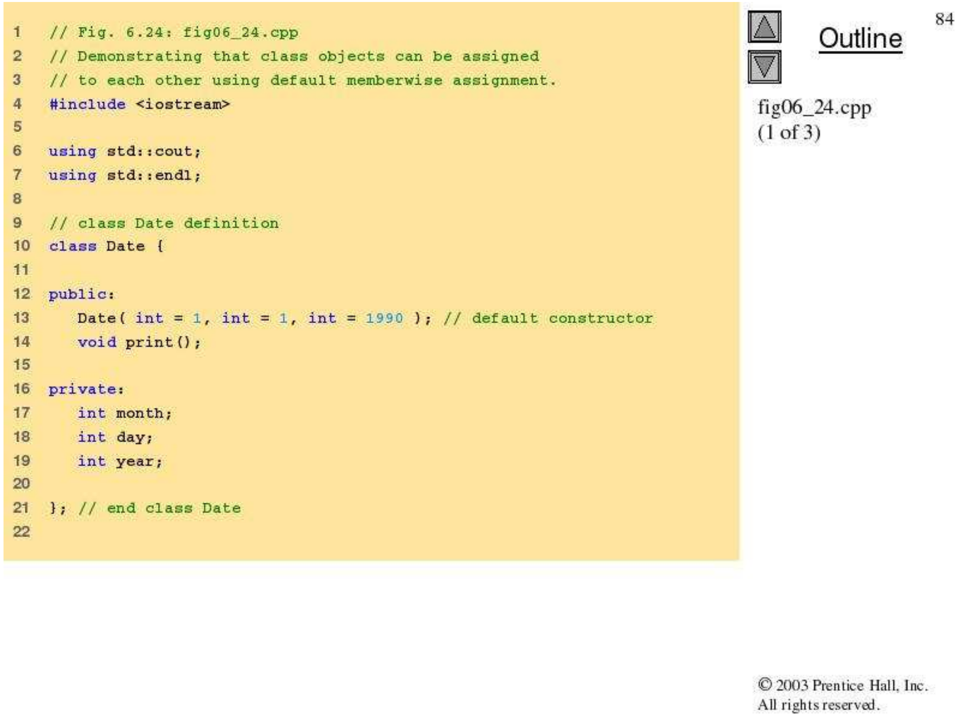
Figure 1.30: *Set* and *get* functions manipulating an object's **private** data. (part 2 of 2)

1.16 Default Memberwise Assignment

The assignment operator (=) can be used to assign an object to another object of the same type.

- 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

Member wise assignment can cause serious problems when used with a class whose data members contain pointers to dynamically allocated storage.

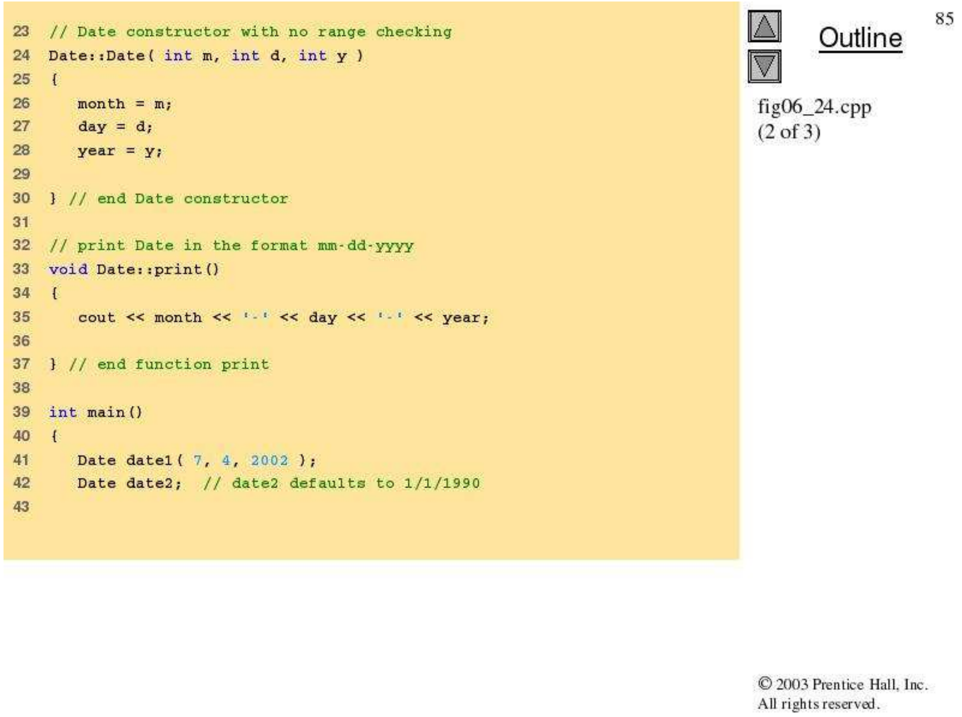


84

Outline

fig06_24.cpp
(1 of 3)

```cpp
// 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

```



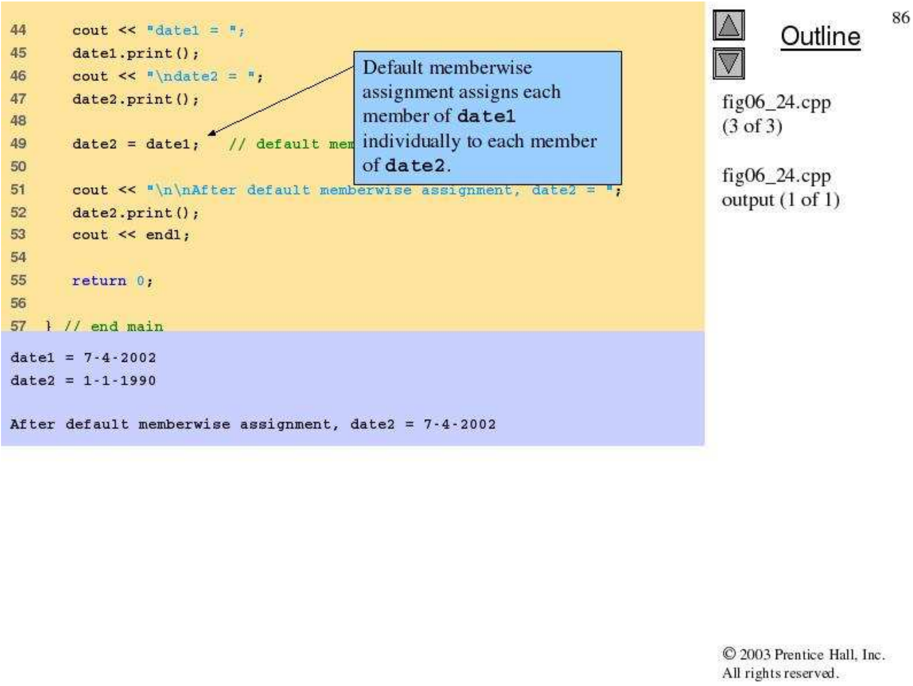
85

Outline

fig06_24.cpp
(2 of 3)

```cpp
// 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

```

Figure 1.31: Default memberwise assignment. (part 1 of 2)



```
    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
```

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

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

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

Figure 1.32: Default memberwise assignment. (part 2 of 2)

1.17 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

Chapter 2

Classes Part II

2.1 const (Constant) Objects and const Member Functions

Some objects need to be modifiable and some do not. The programmer may use keyword **const** to specify that an object is not modifiable and that any attempt to modify the object should result in a compiler error.

- 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 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
- Constructors and destructors

  - Cannot be **const**

  - Must be able to modify objects

    - Constructor; Initializes objects

    - Destructor; Performs termination housekeeping

The program of Figs. 2.1-2.4 modifies class **Time** by making its *get* functions and **printUniversal** function **const.**

- 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

Figs. 2.4-2.6 introduces using *member initializer syntax.* Figs. 2.7-2.8 illustrates the compiler errors for a program that attempts to initialize **const** data member **increment** with an assignment statement in the **Increment** constructor's body rather than with a member initializer.

2.2 Composition: Objects as Members of Classes

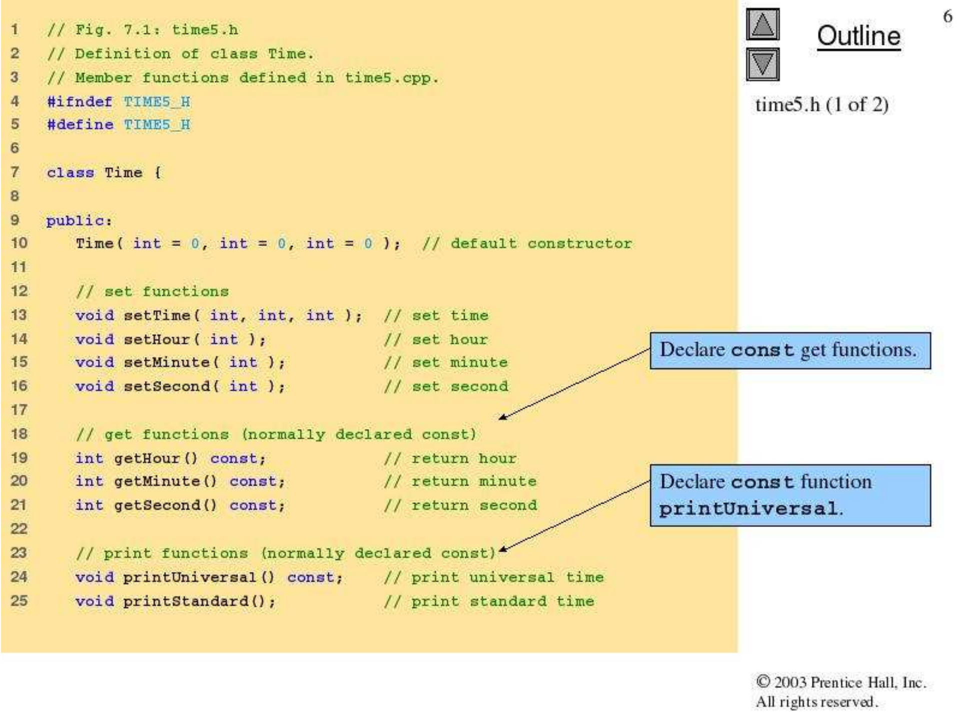
An **AlarmClock** object needs to know when it is supposed to sound its alarm, so why not include a **Time** object as a member of the AlarmClock class? Such a capability is called *composition.*

- 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)

The program of Figs. 2.9-2.14 uses class **Date** and class **Employee** to demonstrate objects as members of other objects. The colon (:) in the header separates the member initializers from the parameter list. In Fig. 2.14, when each of the **Employee**'s **Date** member object's initialized in the **Employee** constructor's member initializer list, the default copy constructor for class **Date** is called. This constructor is defined implicitly by the compiler and does not contain any output statements.

2.3 friend Functions and friend Classes

- friend function
  - Defined outside class's scope
  - Right to access non-public members
- Declaring friends
  - Function; Precede function prototype with keyword **friend**
  - All member functions of class **ClassTwo** as **friend**s of class **ClassOne**
    - Place declaration of form; **friend class ClassTwo;**
    - in **ClassOne** definition
- Properties of friendship
  - Friendship granted, not taken
    - Class **B** **friend** of class **A**; Class **A** must explicitly declare class **B** **friend**

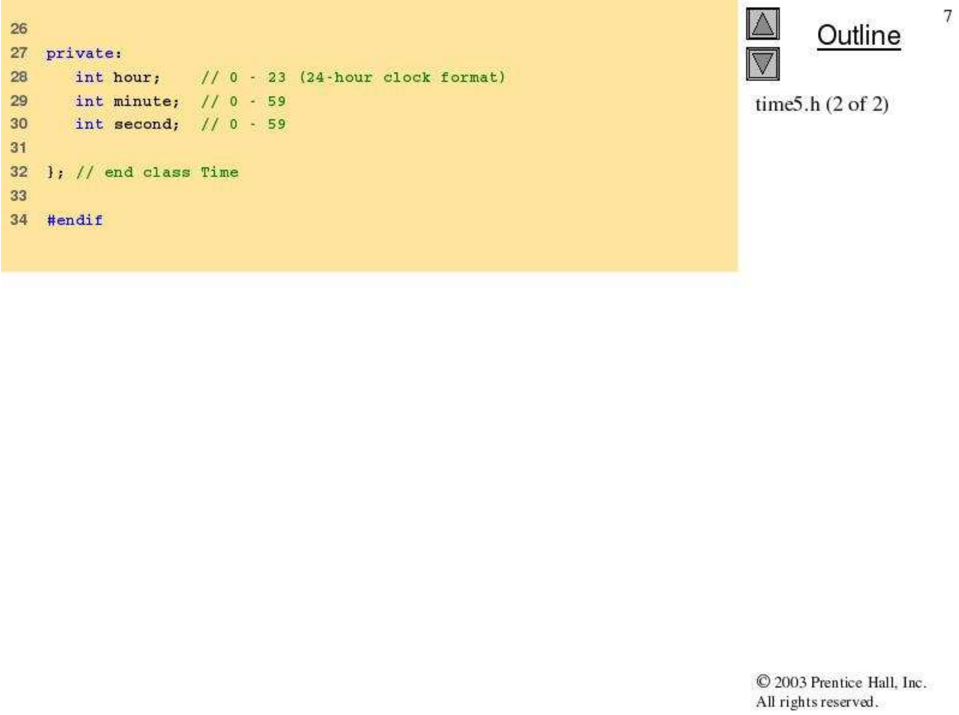


time5.h (1 of 2)

```cpp
// Fig. 7.1: time5.h
// Definition of class Time.
// Member functions defined in time5.cpp.
#ifndef TIME5_H
#define TIME5_H

class Time {

public:
   Time( int = 0, int = 0, int = 0 );  // default constructor

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

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

   // print functions (normally declared const)
   void printUniversal() const;    // print universal time
   void printStandard();           // print standard time
```

Declare **const** get functions.

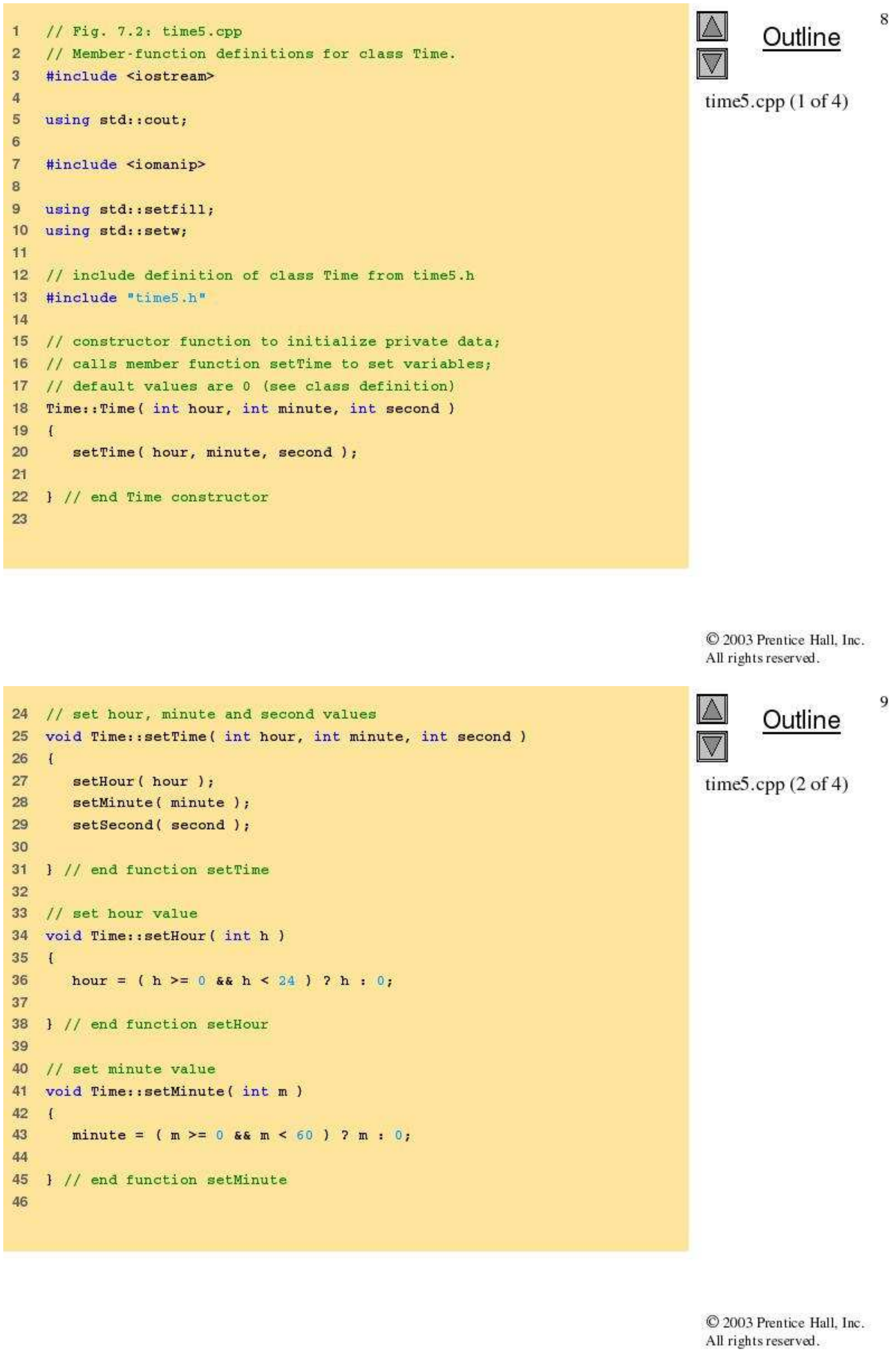
Declare **const** function **printUniversal**.



time5.h (2 of 2)

```cpp

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

}; // end class Time

#endif
```

Figure 2.1: **Time** class definition with **const** member functions.



8

Outline

time5.cpp (1 of 4)

```
// Fig. 7.2: time5.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 time5.h
#include "time5.h"

// constructor function to initialize private data;
// calls member function setTime to set variables;
// default values are 0 (see class definition)
Time::Time( int hour, int minute, int second )
{
   setTime( hour, minute, second );

} // end Time constructor

```

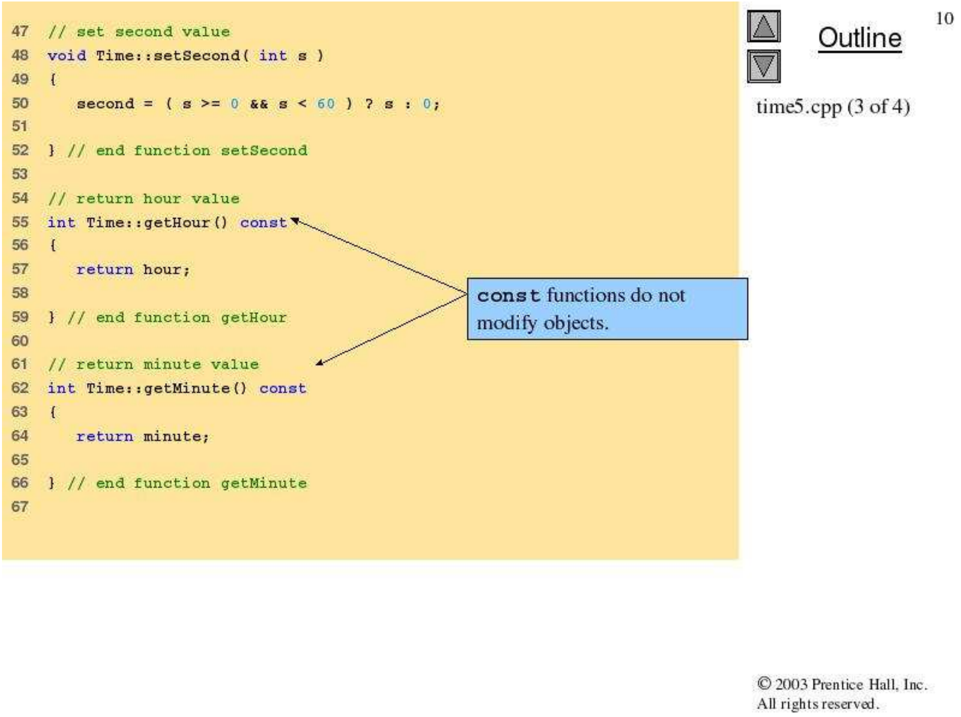
9

Outline

time5.cpp (2 of 4)

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

} // 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

```

Figure 2.2: **Time** class member-function definitions, including **const** member functions. (part 1 of 2)



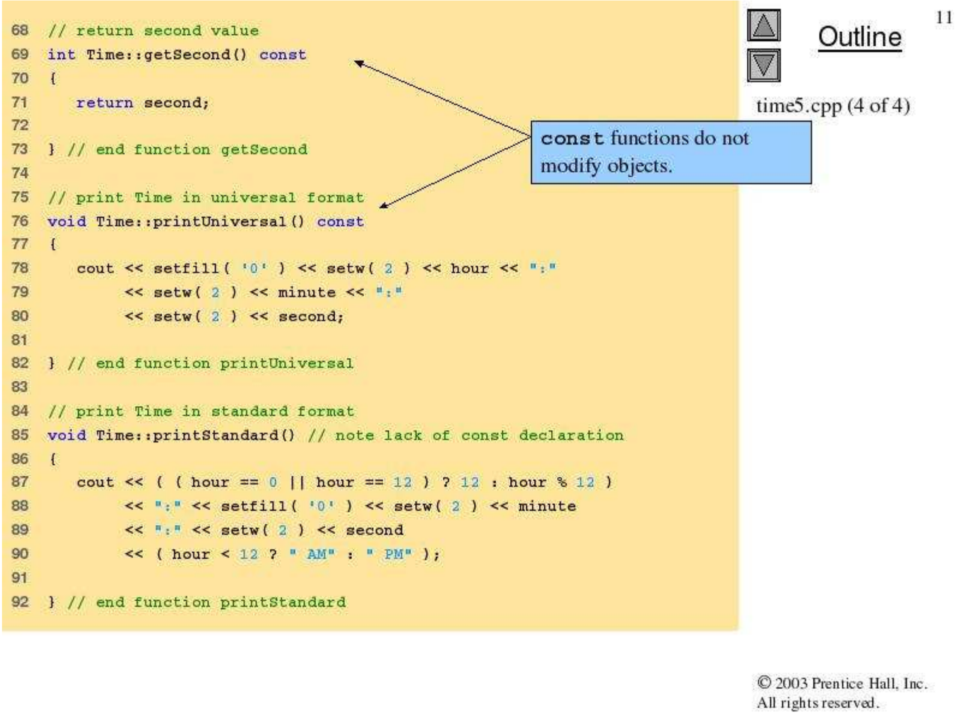
10

Outline

time5.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() const
{
   return hour;

} // end function getHour

// return minute value
int Time::getMinute() const
{
   return minute;

} // end function getMinute

```

const functions do not modify objects.



11

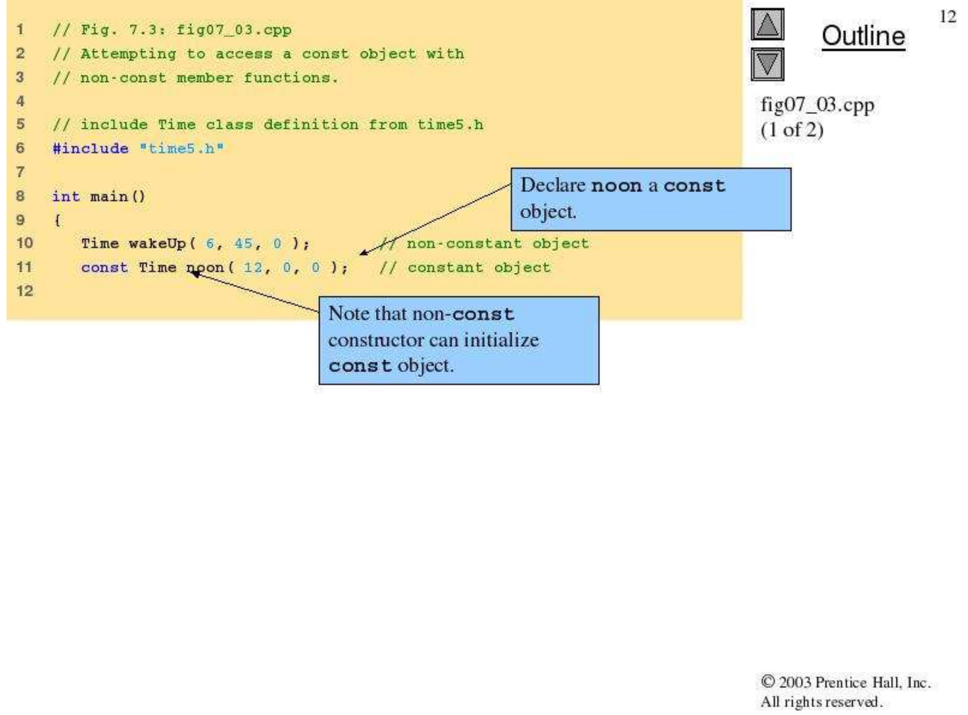
Outline

time5.cpp (4 of 4)

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

} // end function getSecond

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

} // end function printUniversal

// print Time in standard format
void Time::printStandard() // note lack of const declaration
{
   cout << ( ( hour == 0 || hour == 12 ) ? 12 : hour % 12 )
        << ":" << setfill( '0' ) << setw( 2 ) << minute
        << ":" << setw( 2 ) << second
        << ( hour < 12 ? " AM" : " PM" );

} // end function printStandard
```

const functions do not modify objects.

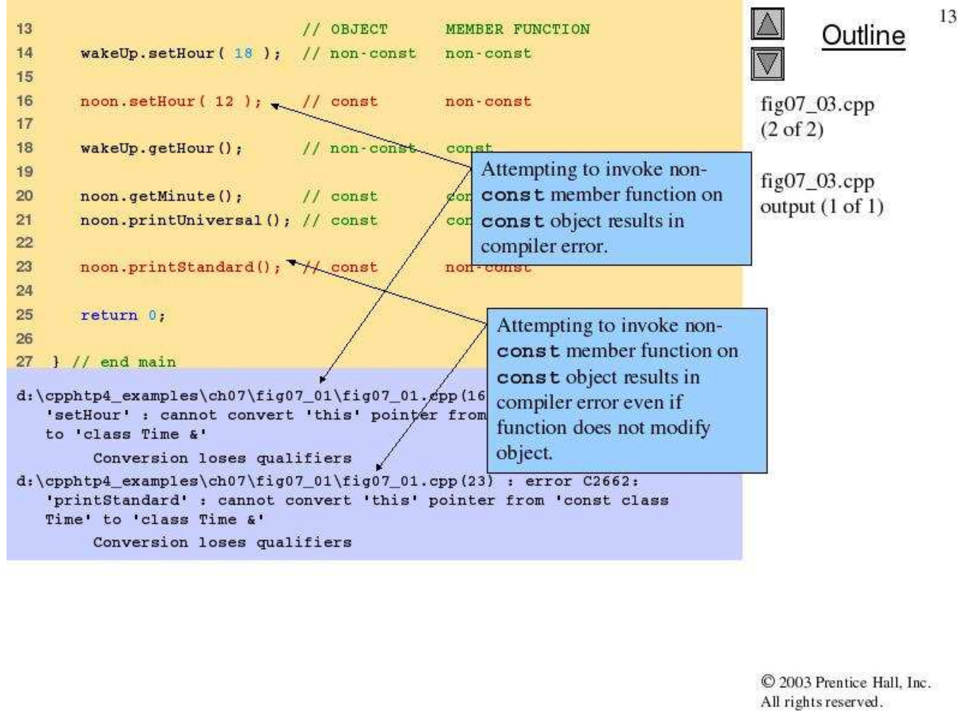
Figure 2.3: **Time** class member-function definitions, including **const** member functions. (part 2 of 2)



```
// Fig. 7.3: fig07_03.cpp
// Attempting to access a const object with
// non-const member functions.

// include Time class definition from time5.h
#include "time5.h"

int main()
{
   Time wakeUp( 6, 45, 0 );      // non-constant object
   const Time noon( 12, 0, 0 );  // constant object

```

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

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



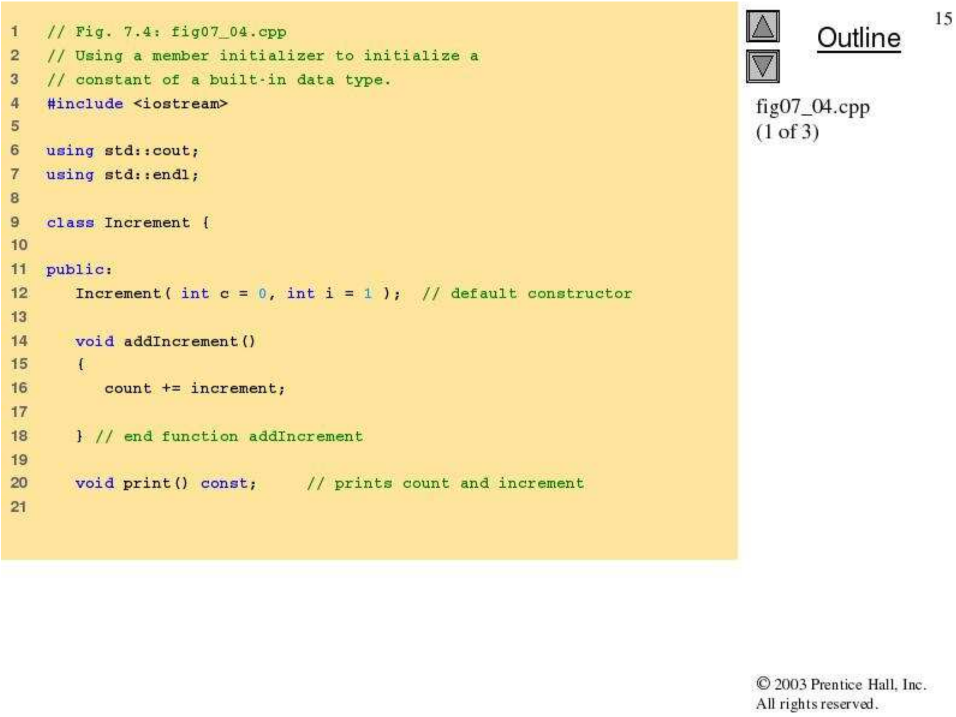
```
                             // OBJECT      MEMBER FUNCTION
   wakeUp.setHour( 18 );     // non-const   non-const

   noon.setHour( 12 );       // const       non-const

   wakeUp.getHour();         // non-const   const

   noon.getMinute();         // const       con
   noon.printUniversal();    // const       con

   noon.printStandard();     // const       non-const

   return 0;

} // end main
```

```
d:\cpphtp4_examples\ch07\fig07_01\fig07_01.cpp(16
   'setHour' : cannot convert 'this' pointer from
   to 'class Time &'
        Conversion loses qualifiers
d:\cpphtp4_examples\ch07\fig07_01\fig07_01.cpp(23) : error C2662:
   'printStandard' : cannot convert 'this' pointer from 'const class
   Time' to 'class Time &'
        Conversion loses qualifiers
```

Attempting to invoke non-**const** member function on **const** object results in compiler error.

Attempting to invoke non-**const** member function on **const** object results in compiler error even if function does not modify object.

Figure 2.4: **const** objects and **const** member functions.



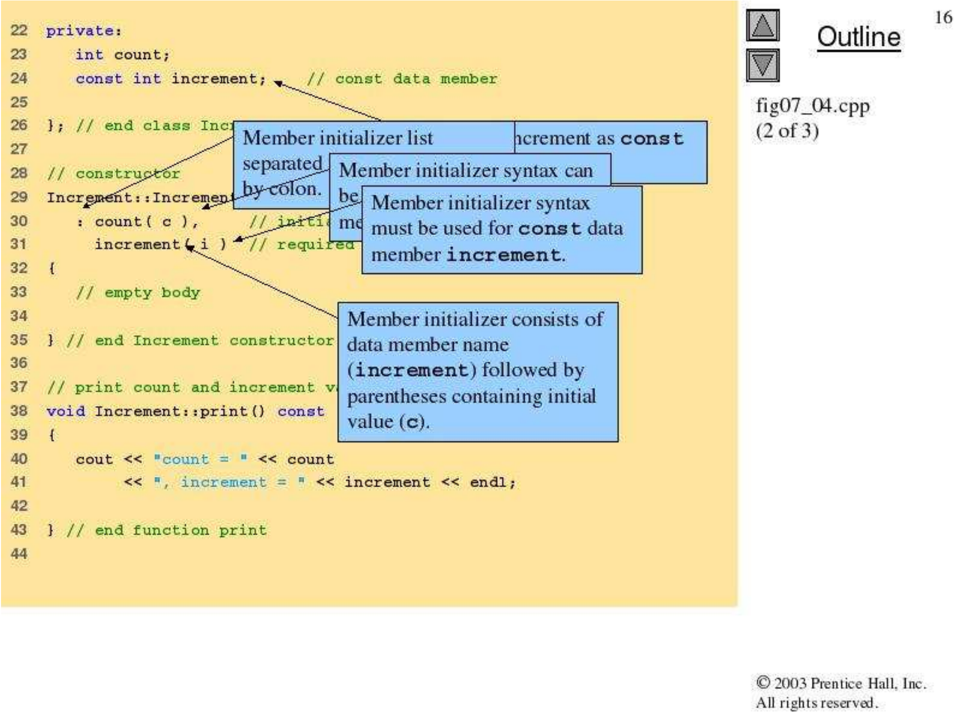
15

Outline

fig07_04.cpp
(1 of 3)

```cpp
// 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

```

© 2003 Prentice Hall, Inc.
All rights reserved.



16

Outline

fig07_04.cpp
(2 of 3)

```cpp
private:
   int count;
   const int increment;    // const data member

}; // end class Inc

// constructor
Increment::Incremen
   : count( c ),      // initi
     increment( i )  // required
{
   // empty body

} // end Increment constructor

// print count and increment v
void Increment::print() const
{
   cout << "count = " << count
        << ", increment = " << increment << endl;

} // end function print

```

Member initializer list separated by colon.

ncrement as **const**

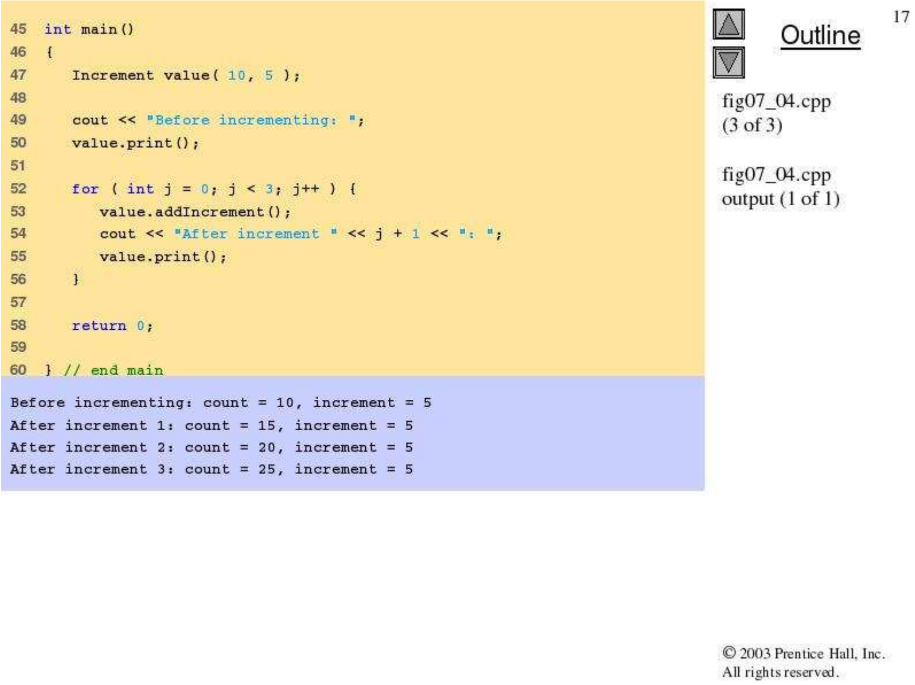
Member initializer syntax can be me

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**).

© 2003 Prentice Hall, Inc.
All rights reserved.

Figure 2.5: Member initializer used to initialize a constant of a built-in data type. (part 1 of 2)



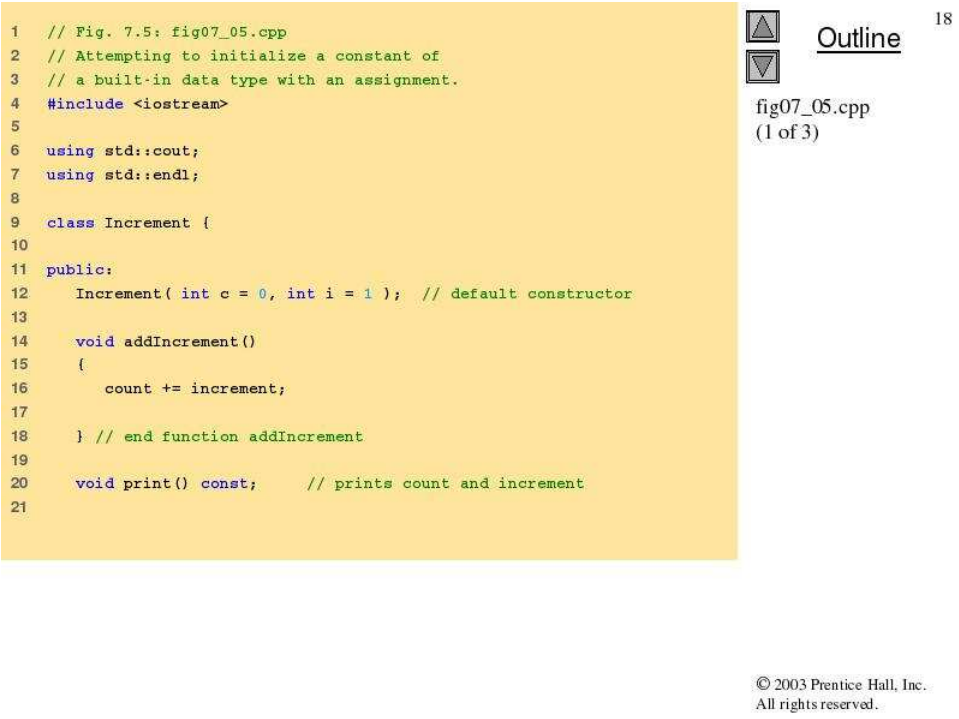
```
int main()
{
   Increment value( 10, 5 );

   cout << "Before incrementing: ";
   value.print();

   for ( int j = 0; j < 3; j++ ) {
      value.addIncrement();
      cout << "After increment " << j + 1 << ": ";
      value.print();
   }

   return 0;

} // 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
```

fig07_04.cpp
(3 of 3)

fig07_04.cpp
output (1 of 1)

Figure 2.6: Member initializer used to initialize a constant of a built-in data type. (part 2 of 2)

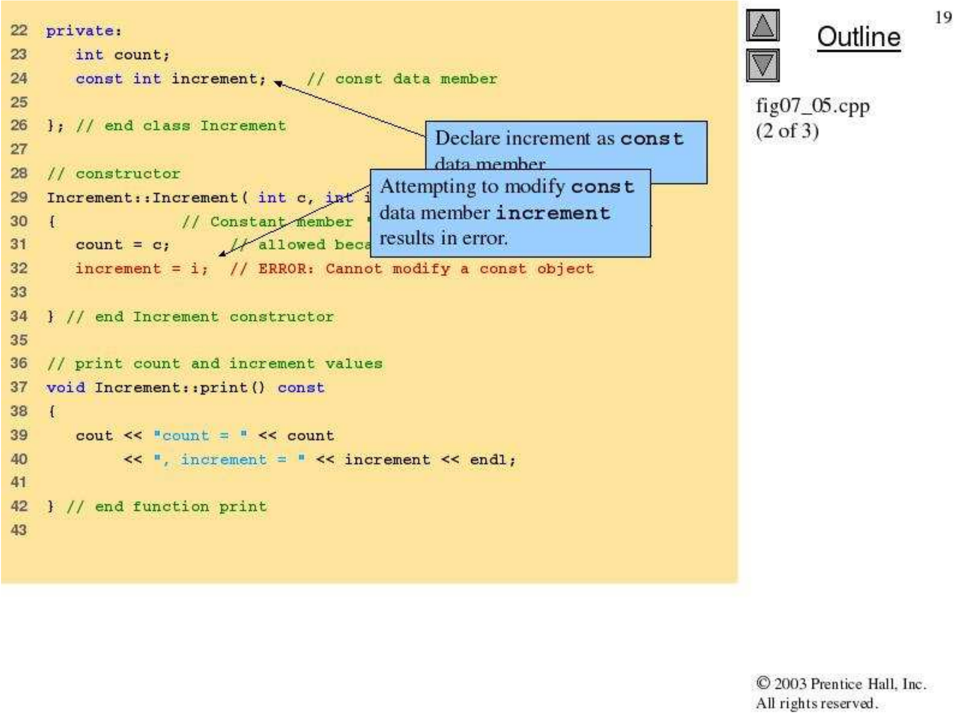


18

Outline

fig07_05.cpp
(1 of 3)

```
// Fig. 7.5: fig07_05.cpp
// Attempting to initialize a constant of
// a built-in data type with an assignment.
#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

```



19

Outline

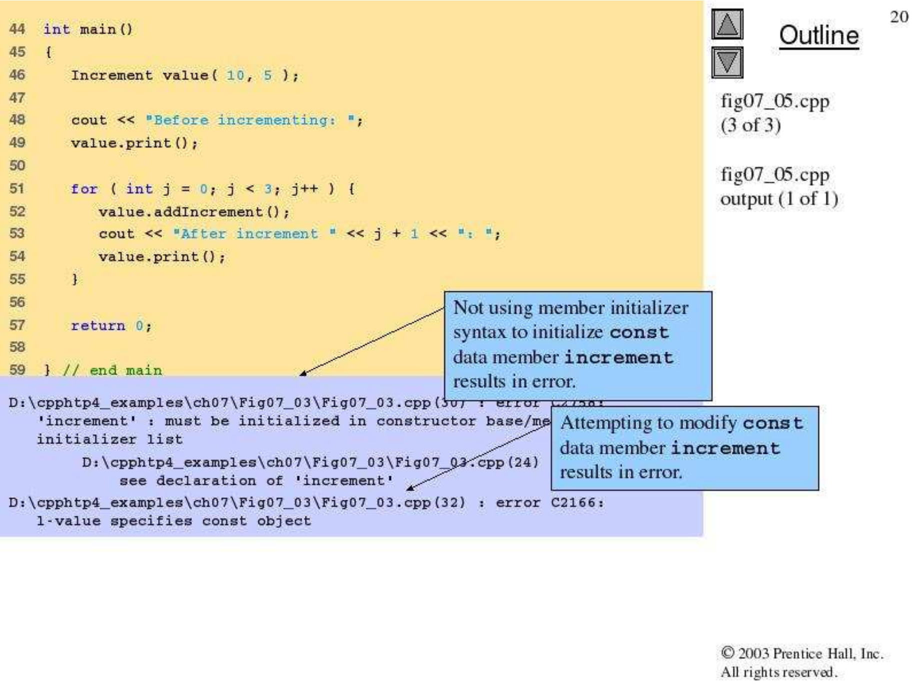
fig07_05.cpp
(2 of 3)

```
private:
   int count;
   const int increment;    // const data member

}; // end class Increment

// constructor
Increment::Increment( int c, int i
{              // Constant member '
   count = c;      // allowed beca
   increment = i;  // ERROR: Cannot modify a const object

} // end Increment constructor

// print count and increment values
void Increment::print() const
{
   cout << "count = " << count
        << ", increment = " << increment << endl;

} // end function print

```

Declare increment as **const** data member

Attempting to modify **const** data member **increment** results in error.

Figure 2.7: Erroneous attempt to initialize a constant of a built-in data type by assignment. (part 1 of 2)



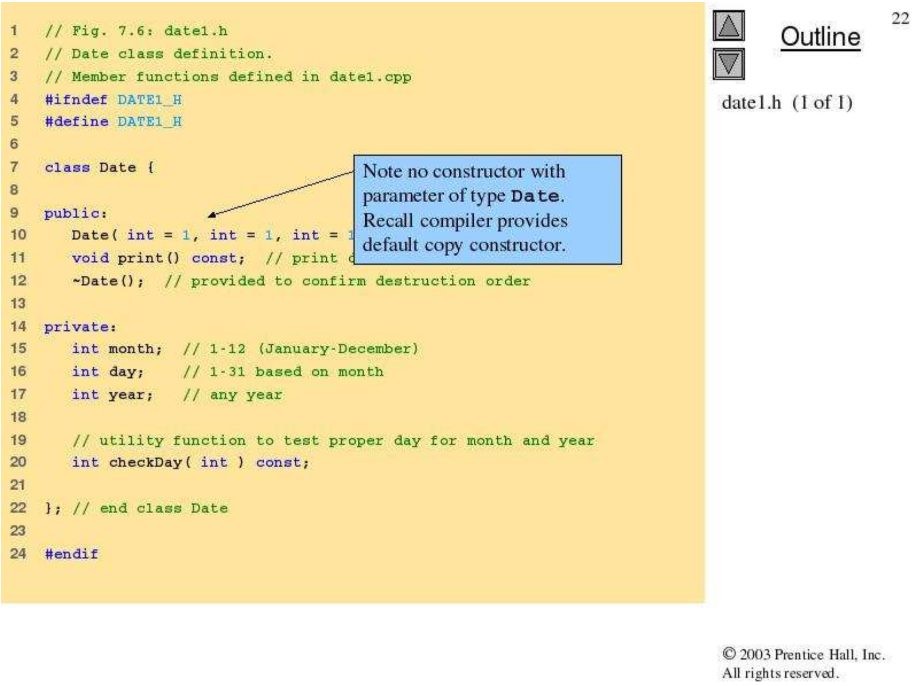
```
int main()
{
   Increment value( 10, 5 );

   cout << "Before incrementing: ";
   value.print();

   for ( int j = 0; j < 3; j++ ) {
      value.addIncrement();
      cout << "After increment " << j + 1 << ": ";
      value.print();
   }

   return 0;

} // end main
```

```
D:\cpphtp4_examples\ch07\Fig07_03\Fig07_03.cpp(30) : error C2758:
   'increment' : must be initialized in constructor base/me
   initializer list
        D:\cpphtp4_examples\ch07\Fig07_03\Fig07_03.cpp(24)
            see declaration of 'increment'
D:\cpphtp4_examples\ch07\Fig07_03\Fig07_03.cpp(32) : error C2166:
   l-value specifies const object
```

Not using member initializer syntax to initialize **const** data member **increment** results in error.

Attempting to modify **const** data member **increment** results in error.

Figure 2.8: Erroneous attempt to initialize a constant of a built-in data type by assignment. (part 2 of 2)



22

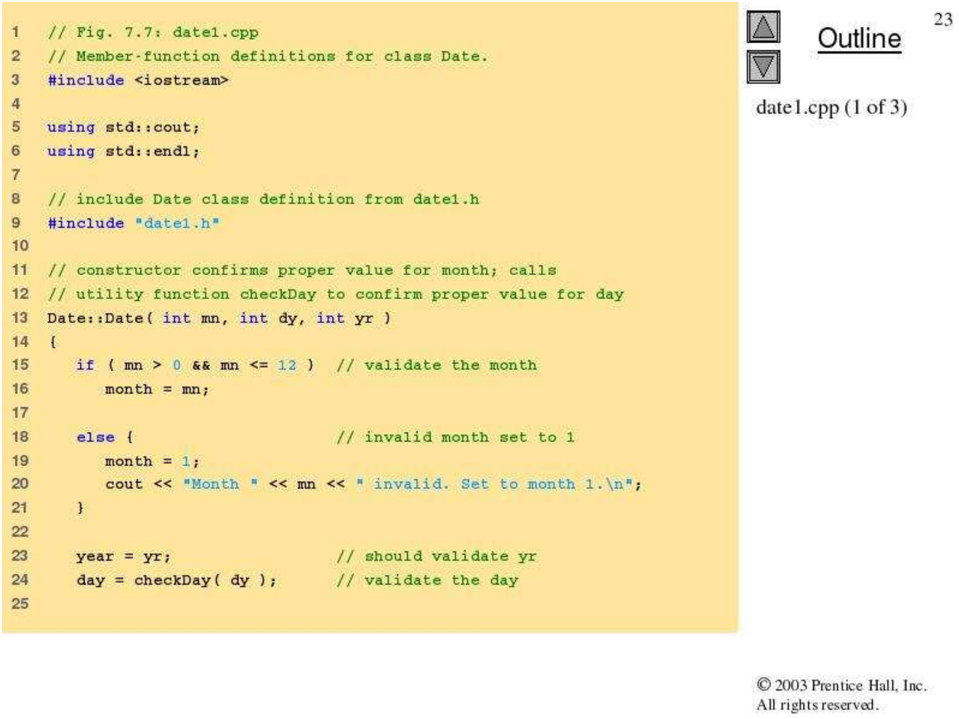
Outline

date1.h (1 of 1)

```
// 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 = 1
   void print() const;  // print d
   ~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.

Figure 2.9: **Date** class definition.

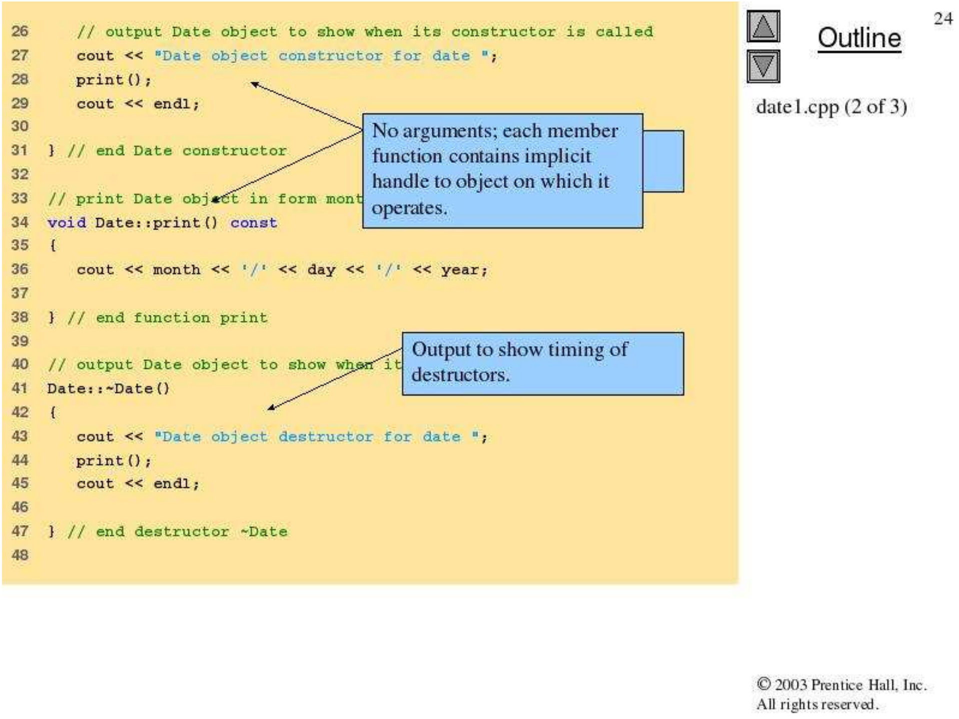


23

Outline

date1.cpp (1 of 3)

```
// Fig. 7.7: date1.cpp
// Member-function definitions for class Date.
#include <iostream>

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

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

// constructor confirms proper value for month; calls
// utility function checkDay to confirm proper value for day
Date::Date( int mn, int dy, int yr )
{
   if ( mn > 0 && mn <= 12 )  // validate the month
      month = mn;

   else {                     // invalid month set to 1
      month = 1;
      cout << "Month " << mn << " invalid. Set to month 1.\n";
   }

   year = yr;                 // should validate yr
   day = checkDay( dy );      // validate the day

```



24

Outline

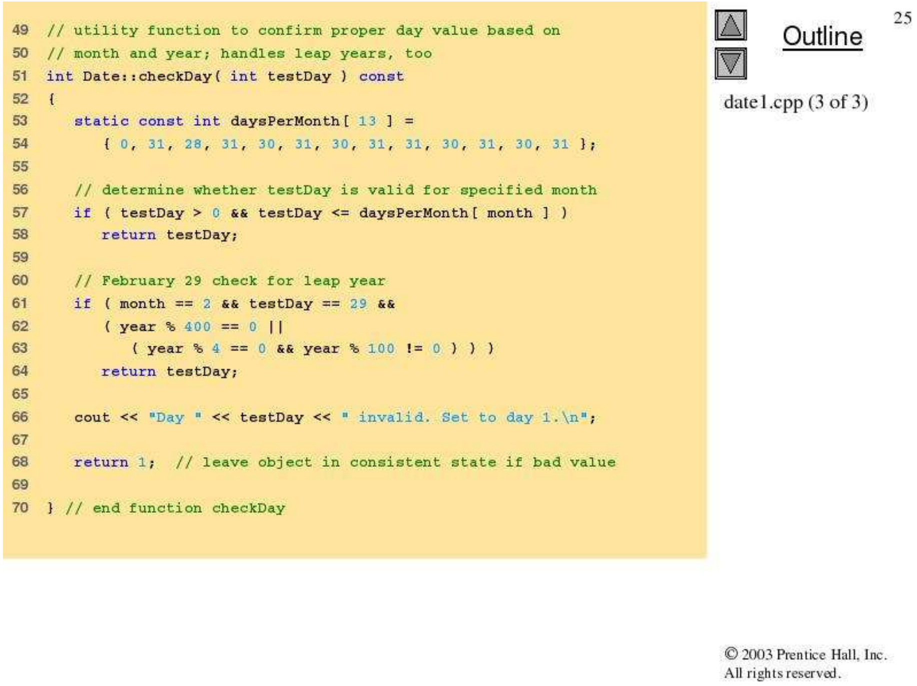
date1.cpp (2 of 3)

```
   // output Date object to show when its constructor is called
   cout << "Date object constructor for date ";
   print();
   cout << endl;

} // end Date constructor

// print Date object in form mont
void Date::print() const
{
   cout << month << '/' << day << '/' << year;

} // end function print

// output Date object to show when it
Date::~Date()
{
   cout << "Date object destructor for date ";
   print();
   cout << endl;

} // end destructor ~Date

```

No arguments; each member function contains implicit handle to object on which it operates.

Output to show timing of destructors.

Figure 2.10: **Date** class member-function definitions. (part 1 of 2)



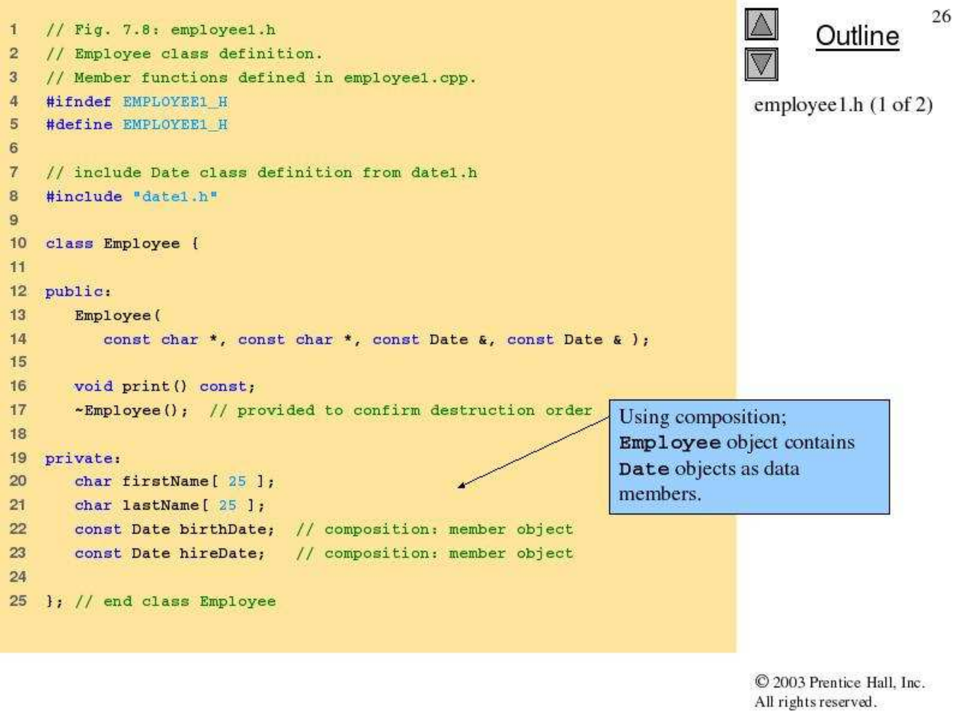
```
// utility function to confirm proper day value based on
// month and year; handles leap years, too
int Date::checkDay( int testDay ) const
{
   static const int daysPerMonth[ 13 ] =
      { 0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 };

   // determine whether testDay is valid for specified month
   if ( testDay > 0 && testDay <= daysPerMonth[ month ] )
      return testDay;

   // February 29 check for leap year
   if ( month == 2 && testDay == 29 &&
      ( year % 400 == 0 ||
         ( year % 4 == 0 && year % 100 != 0 ) ) )
      return testDay;

   cout << "Day " << testDay << " invalid. Set to day 1.\n";

   return 1;  // leave object in consistent state if bad value

} // end function checkDay
```

25

Outline

date1.cpp (3 of 3)

Figure 2.11: **Date** class member-function definitions. (part 2 of 2)



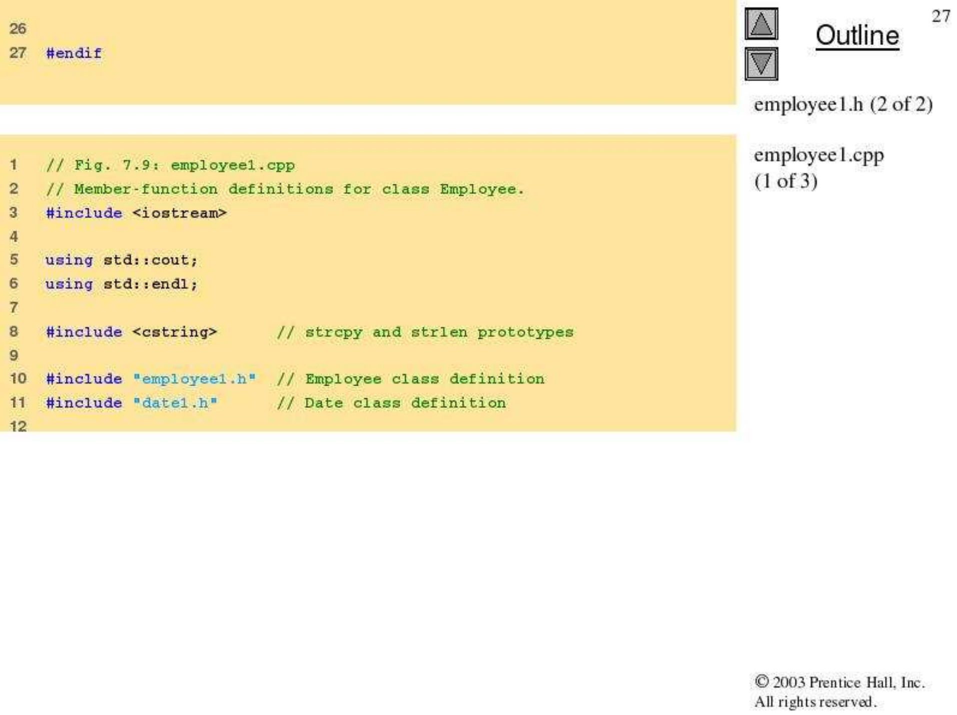
26

Outline

employee1.h (1 of 2)

```
// 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
```

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



27

Outline

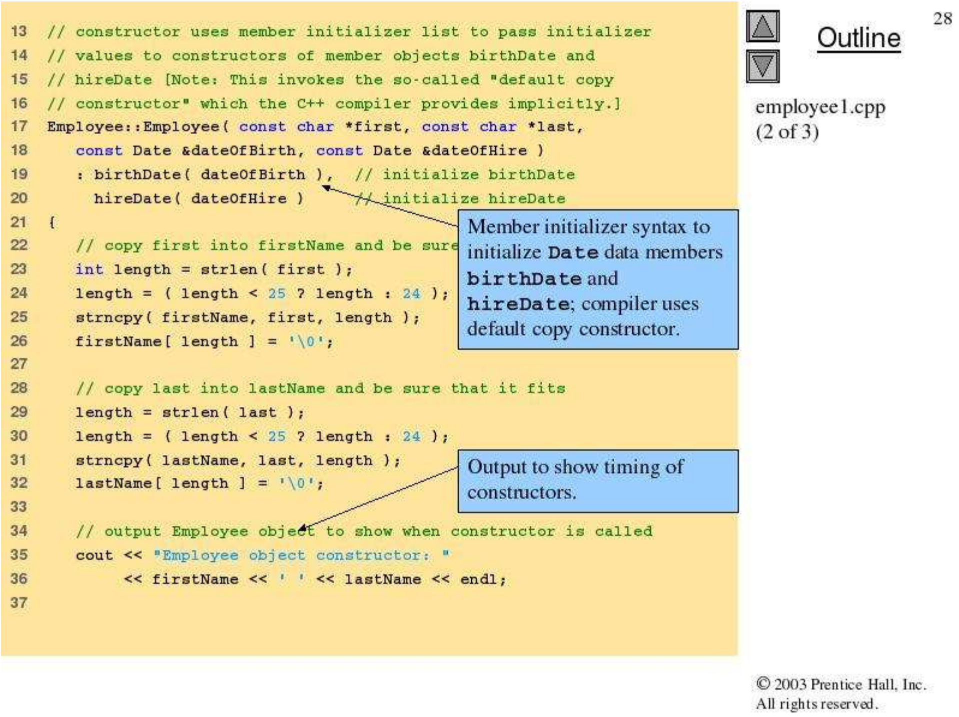
employee1.h (2 of 2)

```

#endif
```

employee1.cpp (1 of 3)

```
// Fig. 7.9: employee1.cpp
// Member-function definitions for class Employee.
#include <iostream>

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

#include <cstring>      // strcpy and strlen prototypes

#include "employee1.h"  // Employee class definition
#include "date1.h"      // Date class definition

```

Figure 2.12: **Employee** class definition showing composition.



28

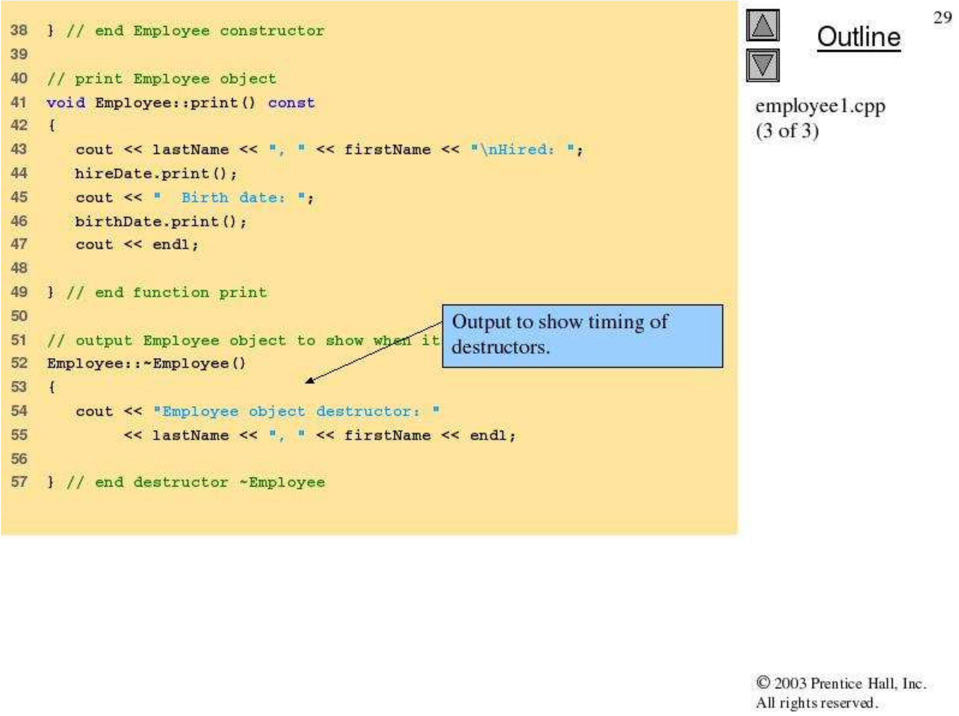
Outline

employee1.cpp
(2 of 3)

```
// 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
   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;

```

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

Output to show timing of constructors.



29

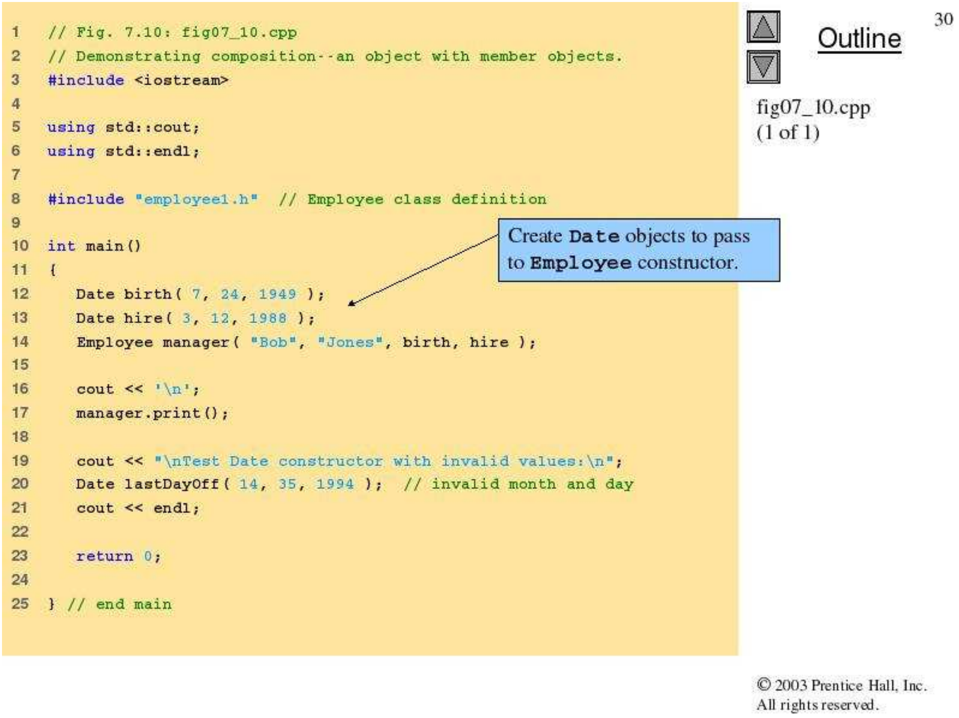
Outline

employee1.cpp
(3 of 3)

```
} // end Employee constructor

// print Employee object
void Employee::print() const
{
   cout << lastName << ", " << firstName << "\nHired: ";
   hireDate.print();
   cout << "  Birth date: ";
   birthDate.print();
   cout << endl;

} // end function print

// output Employee object to show when it
Employee::~Employee()
{
   cout << "Employee object destructor: "
        << lastName << ", " << firstName << endl;

} // end destructor ~Employee
```

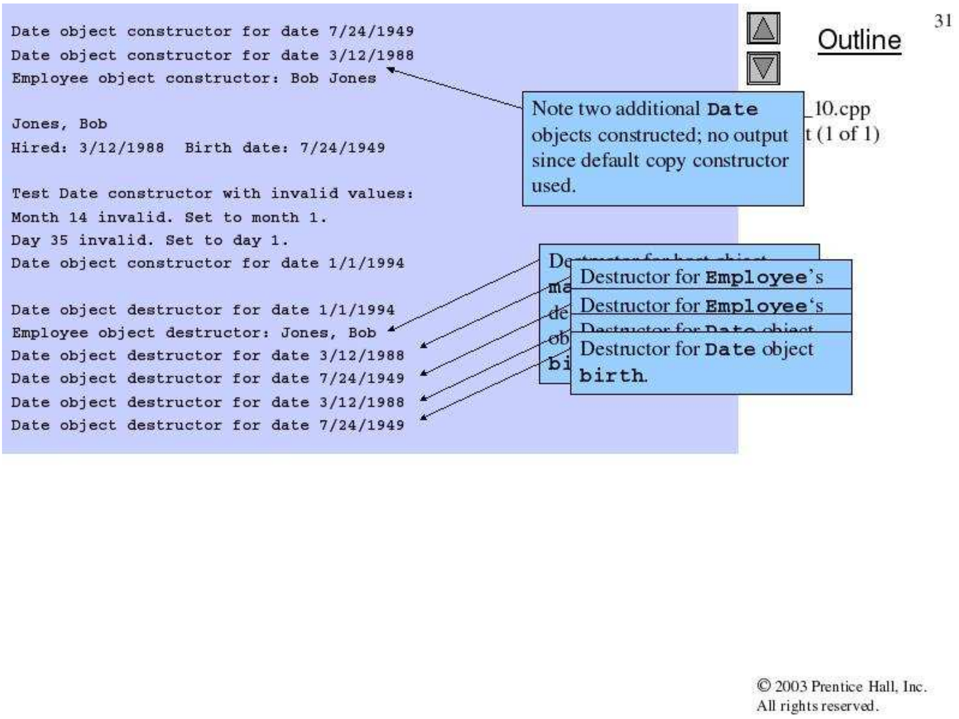
Output to show timing of destructors.

Figure 2.13: **Employee** class member-function definitions,including constructor with a member-initializer list.



```
// Fig. 7.10: fig07_10.cpp
// Demonstrating composition--an object with member objects.
#include <iostream>

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

#include "employee1.h"  // Employee class definition

int main()
{
   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
```

Create **Date** objects to pass to **Employee** constructor.



```
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 **Employee**'s

Destructor for **Employee**'s

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

Figure 2.14: Member-object initializers.

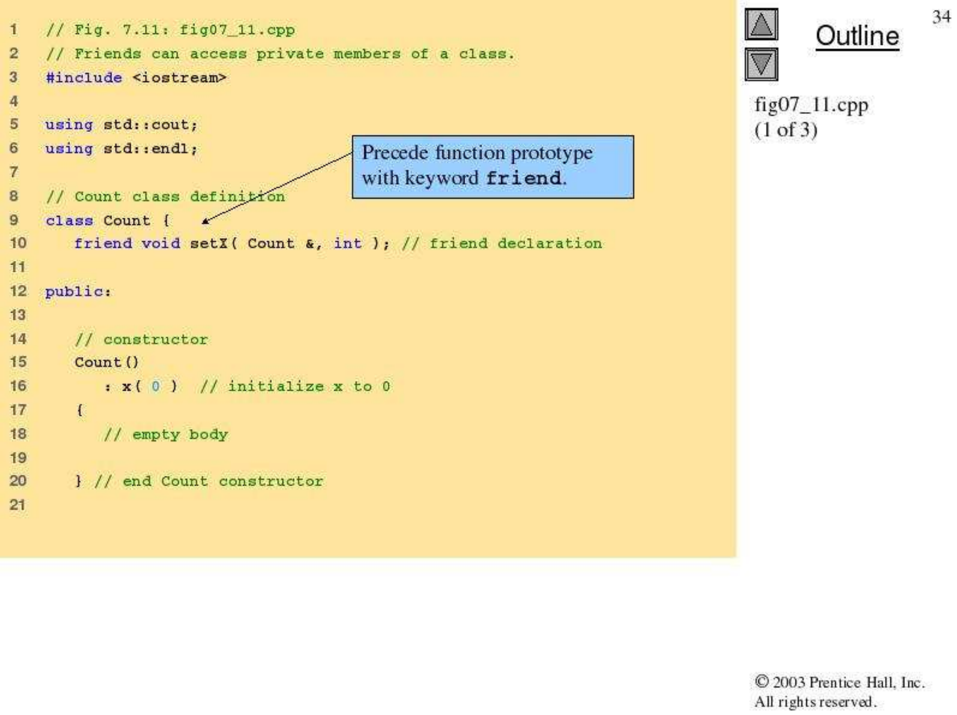
- Not symmetric
  - Class **B** **friend** of class **A**
  - Class **A** not necessarily **friend** of class **B**
- Not transitive
  - Class **A** **friend** of class **B**
  - Class **B** **friend** of class **C**
  - Class **A** not necessarily **friend** of Class **C**

The program of Figs. 2.15-2.16 (top) defines friend function **setX** to set the **private** data member **x** of class **Count**. Friend declaration can appear anywhere in the class. The program of Figs. 2.16 (bottom) -2.17 demonstrates the error messages produced by the compiler when nonfriend function **cannotSetX** is called to modify **private** data member **x**.

2.4 Using the this Pointer

We have seen that an object's member functions can manipulate the object's data. How do member functions know which object's data members to manipulate? Every object has access to its own address through a pointer called **this** (a C++ keyword).

- Allows object to access own address
- Not part of object itself; Implicit argument to non-**static** member function call
- Implicitly reference member data and functions
- Type of **this** pointer depends on
  - Type of object
  - Whether member function is **const**
  - In non-**const** member function of **Employee**
    - **this** has type **Employee * const** ; Constant pointer to non-constant **Employee** object
  - In **const** member function of **Employee**
    - **this** has type **const Employee * const** ; Constant pointer to constant **Employee** object



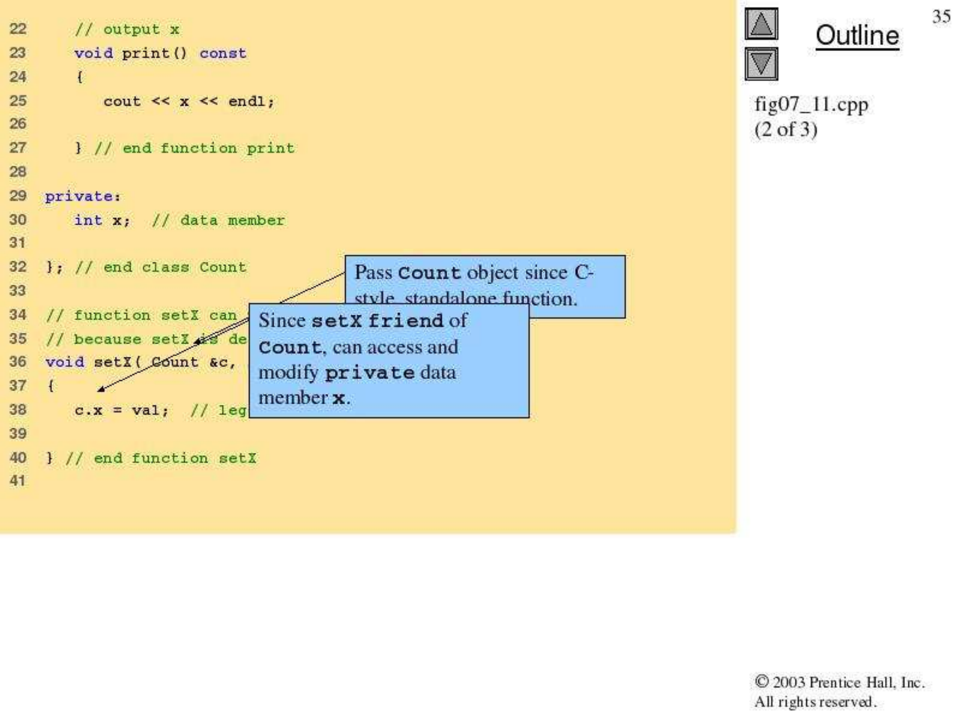
34

Outline

fig07_11.cpp
(1 of 3)

```
// Fig. 7.11: fig07_11.cpp
// Friends can access private members of a class.
#include <iostream>

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

// Count class definition
class Count {
   friend void setX( Count &, int ); // friend declaration

public:

   // constructor
   Count()
      : x( 0 )  // initialize x to 0
   {
      // empty body

   } // end Count constructor

```

Precede function prototype with keyword **friend**.



35

Outline

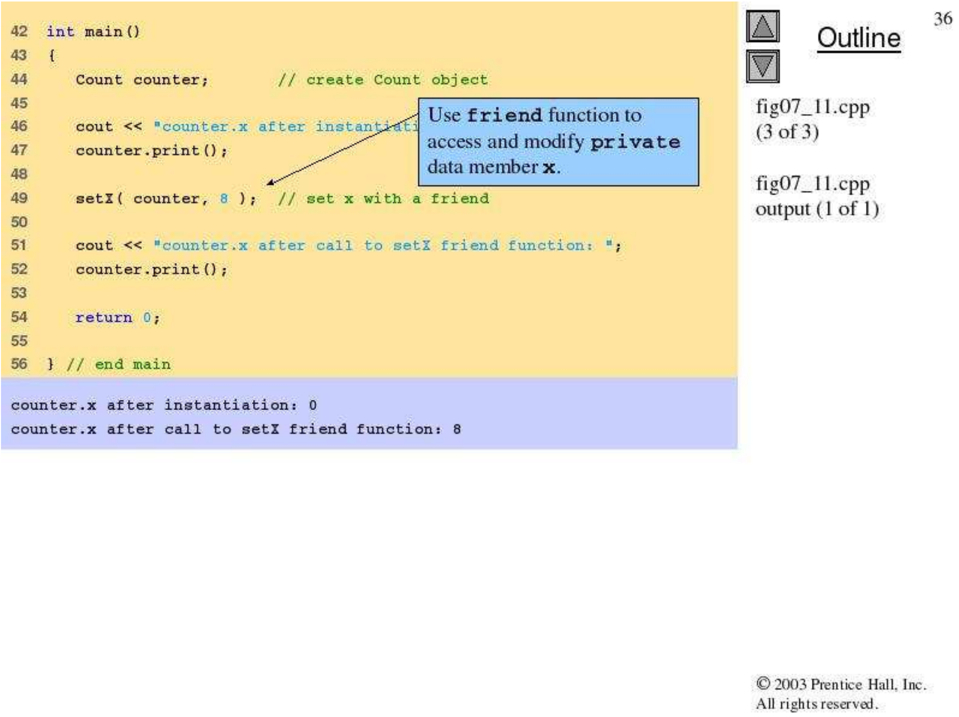
fig07_11.cpp
(2 of 3)

```
   // output x
   void print() const
   {
      cout << x << endl;

   } // end function print

private:
   int x;  // data member

}; // end class Count

// function setX can
// because setX is de
void setX( Count &c,
{
   c.x = val;  // leg

} // end function setX

```

Pass **Count** object since C-style, standalone function.

Since **setX friend** of **Count**, can access and modify **private** data member **x**.

Figure 2.15: Friends can access **private** members of the class.



```
int main()
{
   Count counter;       // create Count object

   cout << "counter.x after instantiati
   counter.print();

   setX( counter, 8 );  // set x with a friend

   cout << "counter.x after call to setX friend function: ";
   counter.print();

   return 0;

} // end main
```

```
counter.x after instantiation: 0
counter.x after call to setX friend function: 8
```

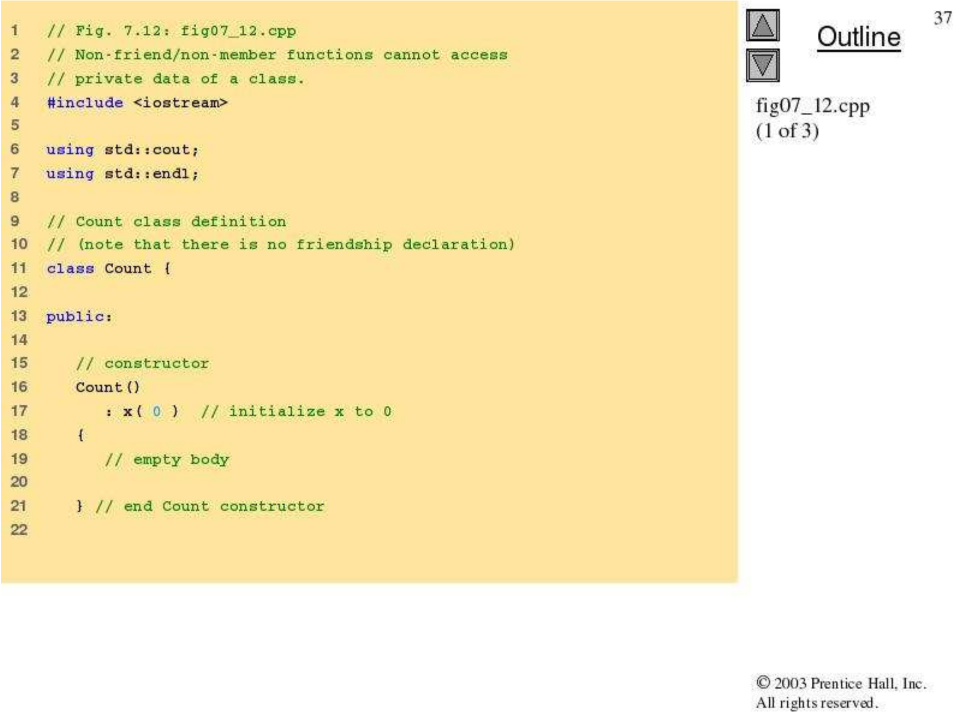
Use **friend** function to access and modify **private** data member **x**.

36

Outline

fig07_11.cpp
(3 of 3)

fig07_11.cpp
output (1 of 1)



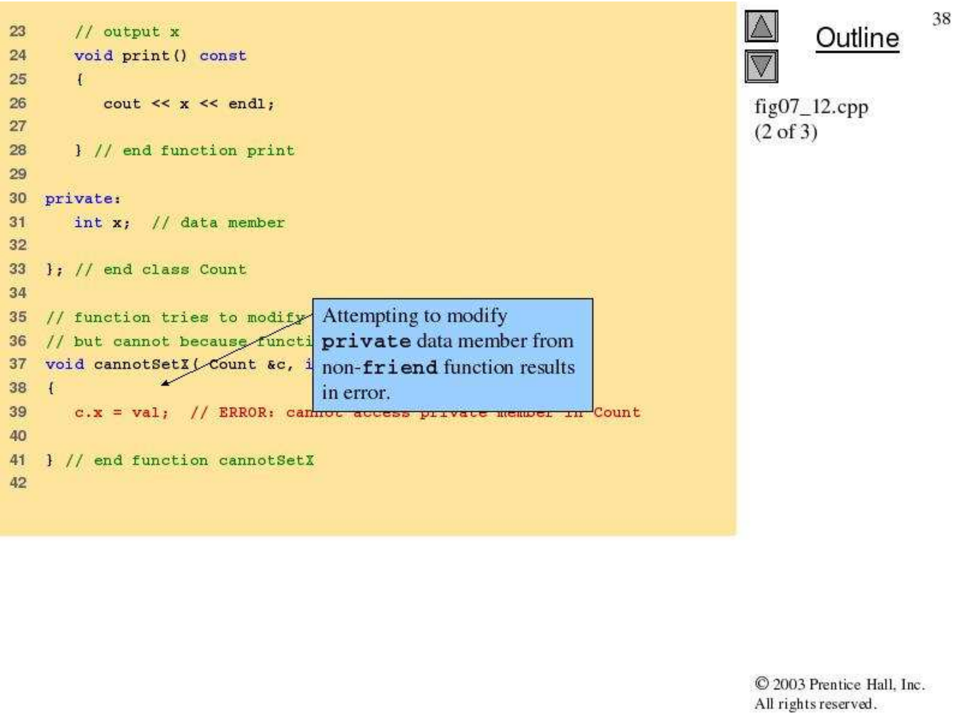
```
// Fig. 7.12: fig07_12.cpp
// Non-friend/non-member functions cannot access
// private data of a class.
#include <iostream>

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

// Count class definition
// (note that there is no friendship declaration)
class Count {

public:

   // constructor
   Count()
      : x( 0 )  // initialize x to 0
   {
      // empty body

   } // end Count constructor

```

37

Outline

fig07_12.cpp
(1 of 3)

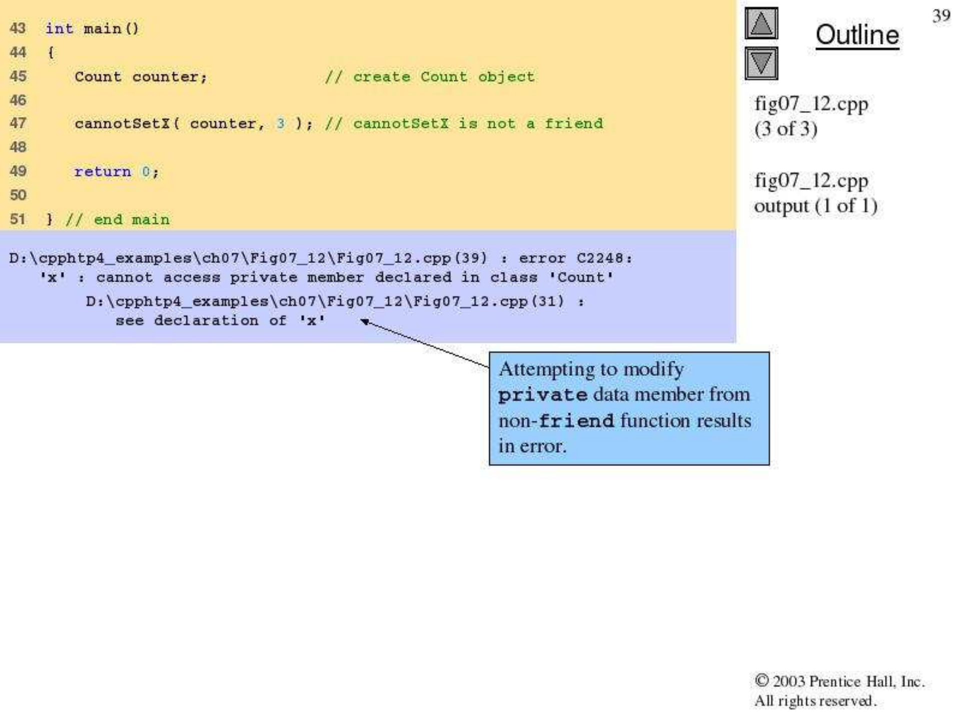
Figure 2.16: Nonfriend/nonmember functions cannot access **private** members. (part 1 of 2)



```
23     // output x
24     void print() const
25     {
26        cout << x << endl;
27
28     } // end function print
29
30  private:
31     int x;  // data member
32
33  }; // end class Count
34
35  // function tries to modify
36  // but cannot because functi
37  void cannotSetX( Count &c, i
38  {
39     c.x = val;  // ERROR: cannot access private member in Count
40
41  } // end function cannotSetX
42
```

Attempting to modify **`private`** data member from non-**`friend`** function results in error.

fig07_12.cpp
(2 of 3)



```
43  int main()
44  {
45     Count counter;             // create Count object
46
47     cannotSetX( counter, 3 ); // cannotSetX is not a friend
48
49     return 0;
50
51  } // end main
```

```
D:\cpphtp4_examples\ch07\Fig07_12\Fig07_12.cpp(39) : error C2248:
   'x' : cannot access private member declared in class 'Count'
        D:\cpphtp4_examples\ch07\Fig07_12\Fig07_12.cpp(31) :
           see declaration of 'x'
```

Attempting to modify **`private`** data member from non-**`friend`** function results in error.

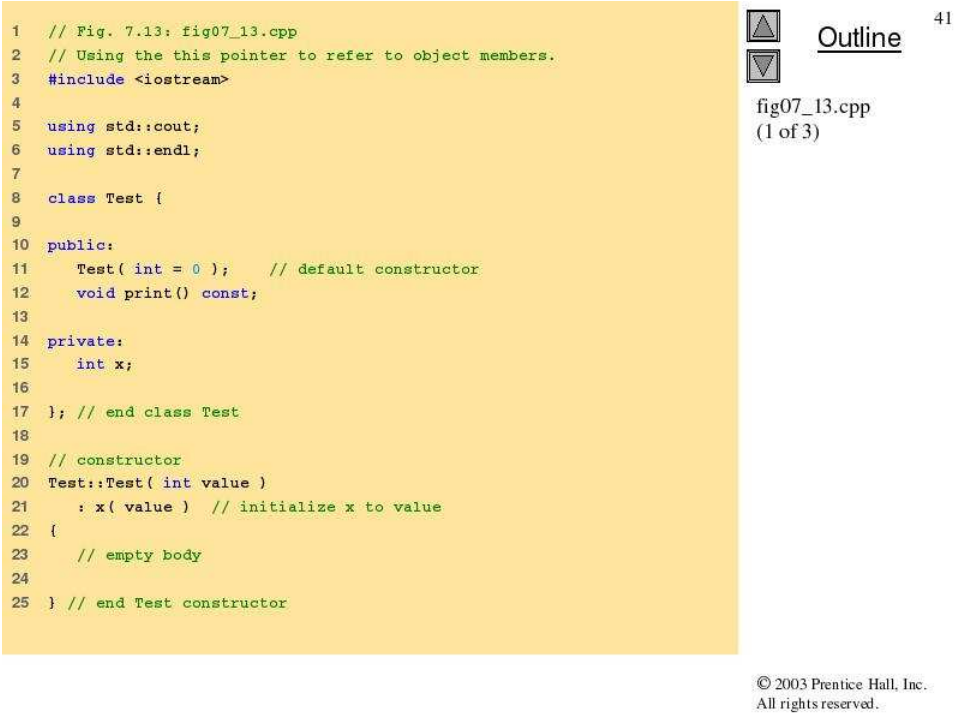
fig07_12.cpp
(3 of 3)

fig07_12.cpp
output (1 of 1)

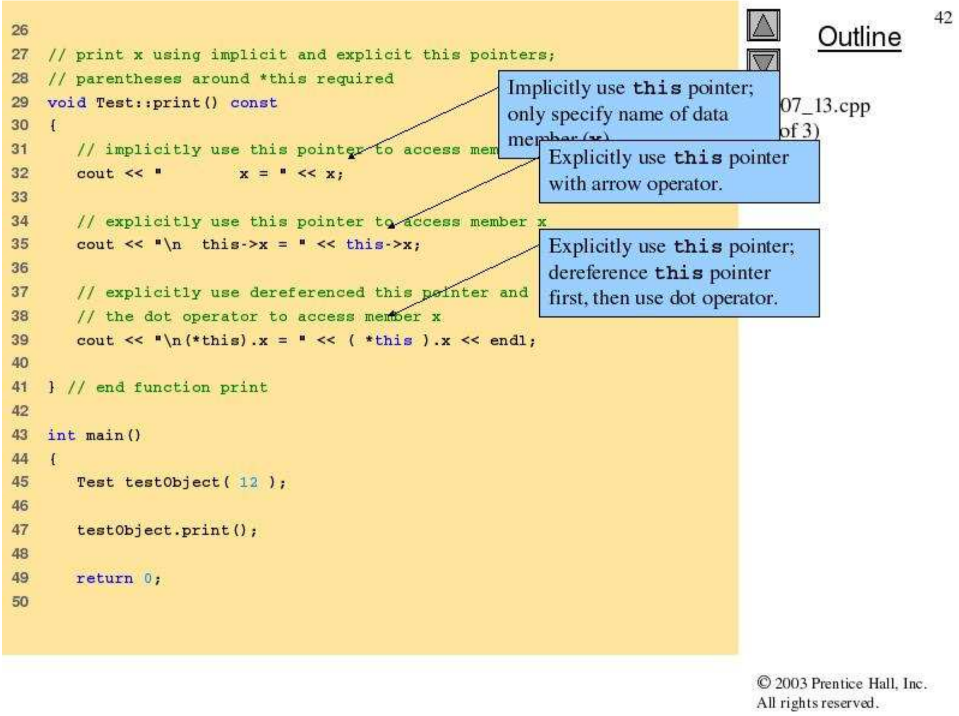
Figure 2.17: Nonfriend/nonmember functions cannot access **private** members. (part 2 of 2)

The program of Figs. 2.18-2.19 demonstrates the implicit and explicit use of the **this** pointer to enable a member function of class **Test** to print the **private** data **x** of a **Test** object. The program of Figs. 2.20-2.24 modifies class **Time**'s *set* functions **setTime, setHour, setMinute** and **setSecond** such that each returns a reference to a **Time** object to enable cascaded member-function calls.

- Cascaded member function calls
  - Multiple functions invoked in same statement
  - Function returns reference pointer to same object; **{ return *this; }**
- Other functions operate on that pointer
- Functions that do not return references must be called last



```
// Fig. 7.13: fig07_13.cpp
// Using the this pointer to refer to object members.
#include <iostream>

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

class Test {

public:
   Test( int = 0 );     // default constructor
   void print() const;

private:
   int x;

}; // end class Test

// constructor
Test::Test( int value )
   : x( value )  // initialize x to value
{
   // empty body

} // end Test constructor
```



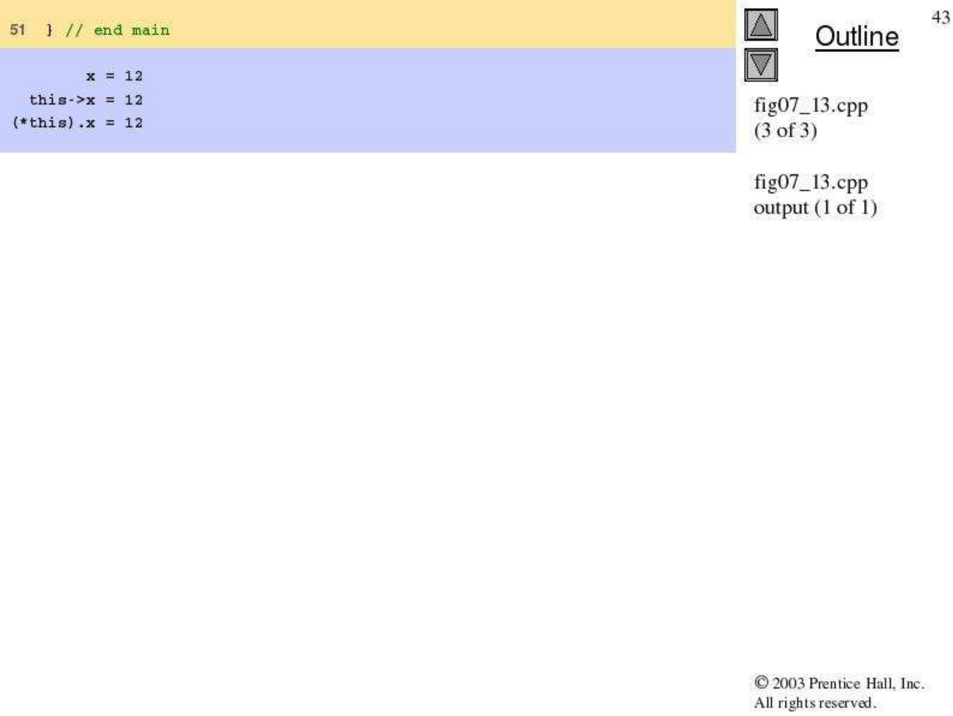
```

// print x using implicit and explicit this pointers;
// parentheses around *this required
void Test::print() const
{
   // implicitly use this pointer to access mem
   cout << "        x = " << x;

   // explicitly use this pointer to access member x
   cout << "\n  this->x = " << this->x;

   // explicitly use dereferenced this pointer and
   // the dot operator to access member x
   cout << "\n(*this).x = " << ( *this ).x << endl;

} // end function print

int main()
{
   Test testObject( 12 );

   testObject.print();

   return 0;

```

Implicitly use **this** pointer; only specify name of data member (x).

Explicitly use **this** pointer with arrow operator.

Explicitly use **this** pointer; dereference **this** pointer first, then use dot operator.

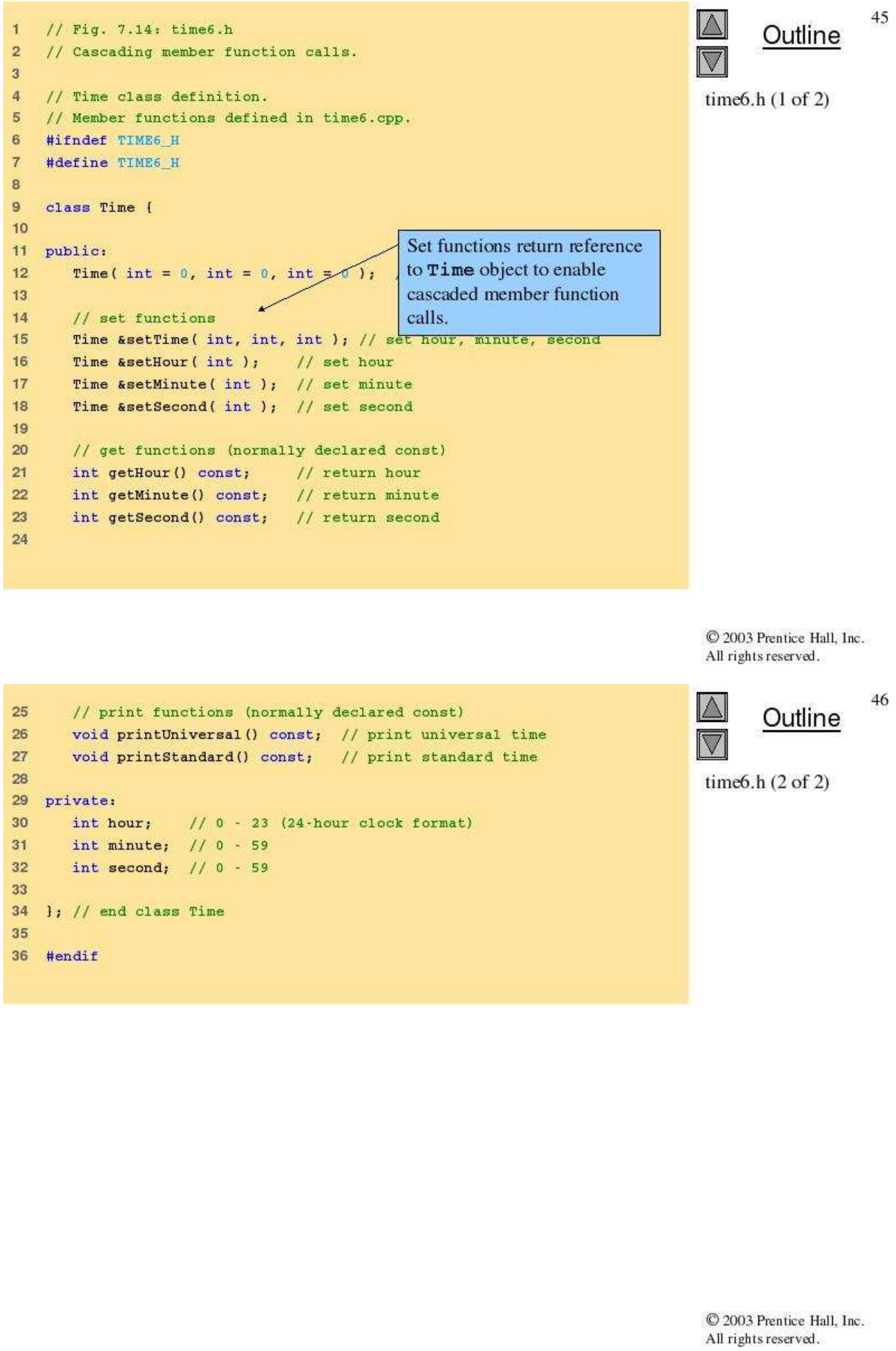
Figure 2.18: **this** pointer implicitly and explicitly used to access an object's members. (part 1 of 2)



```
} // end main
```

```
        x = 12
  this->x = 12
(*this).x = 12
```

Figure 2.19: **this** pointer implicitly and explicitly used to access an object's members. (part 2 of 2)

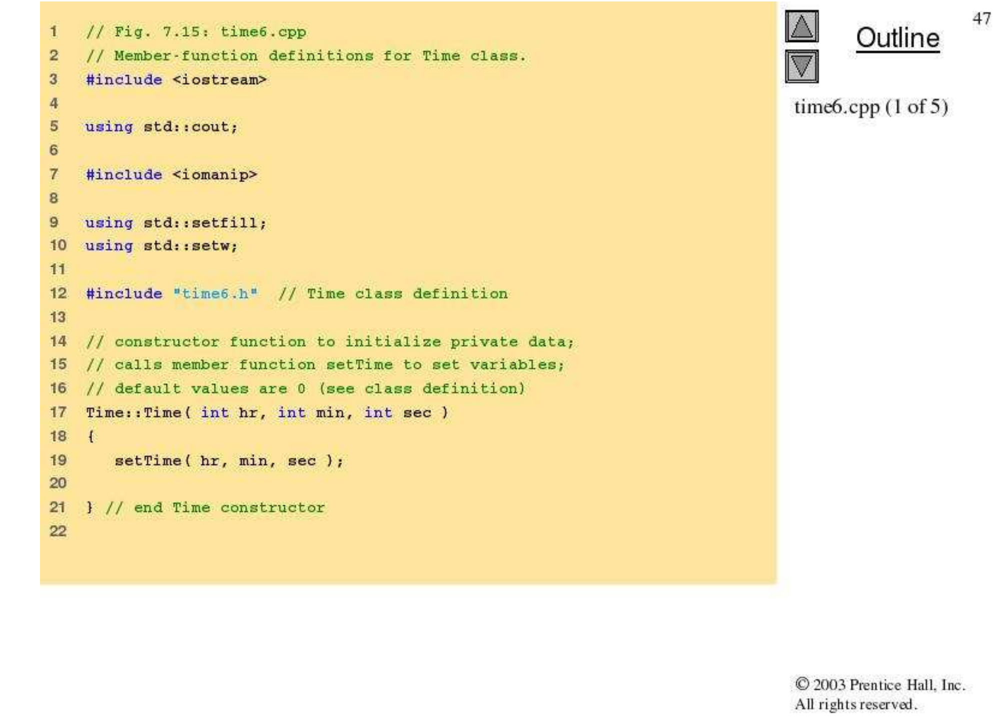


```
// Fig. 7.14: time6.h
// Cascading member function calls.

// Time class definition.
// Member functions defined in time6.cpp.
#ifndef TIME6_H
#define TIME6_H

class Time {

public:
   Time( int = 0, int = 0, int = 0 );

   // set functions
   Time &setTime( int, int, int ); // set hour, minute, second
   Time &setHour( int );     // set hour
   Time &setMinute( int );   // set minute
   Time &setSecond( int );   // set second

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

```

Set functions return reference to **Time** object to enable cascaded member function calls.

```
   // print functions (normally declared const)
   void printUniversal() const;  // print universal time
   void printStandard() const;   // print standard time

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

}; // end class Time

#endif
```

Figure 2.20: **Time** class definition modified to enable cascaded member-function calls.



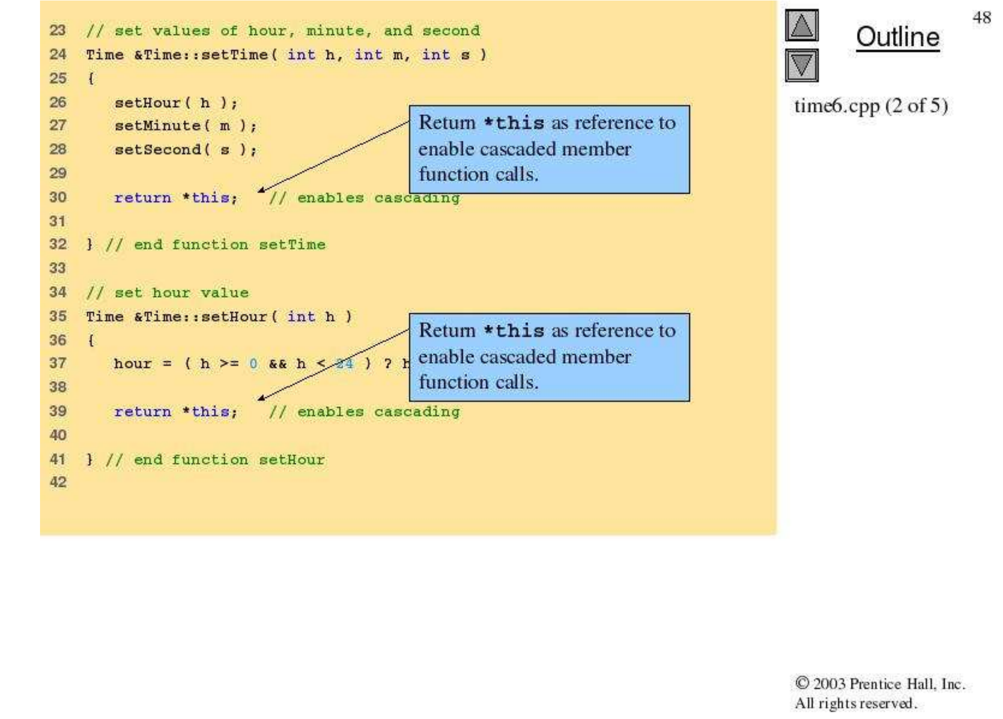
47

Outline

time6.cpp (1 of 5)

```
// Fig. 7.15: time6.cpp
// Member-function definitions for Time class.
#include <iostream>

using std::cout;

#include <iomanip>

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

#include "time6.h"  // Time class definition

// 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

```

© 2003 Prentice Hall, Inc.
All rights reserved.



48

Outline

time6.cpp (2 of 5)

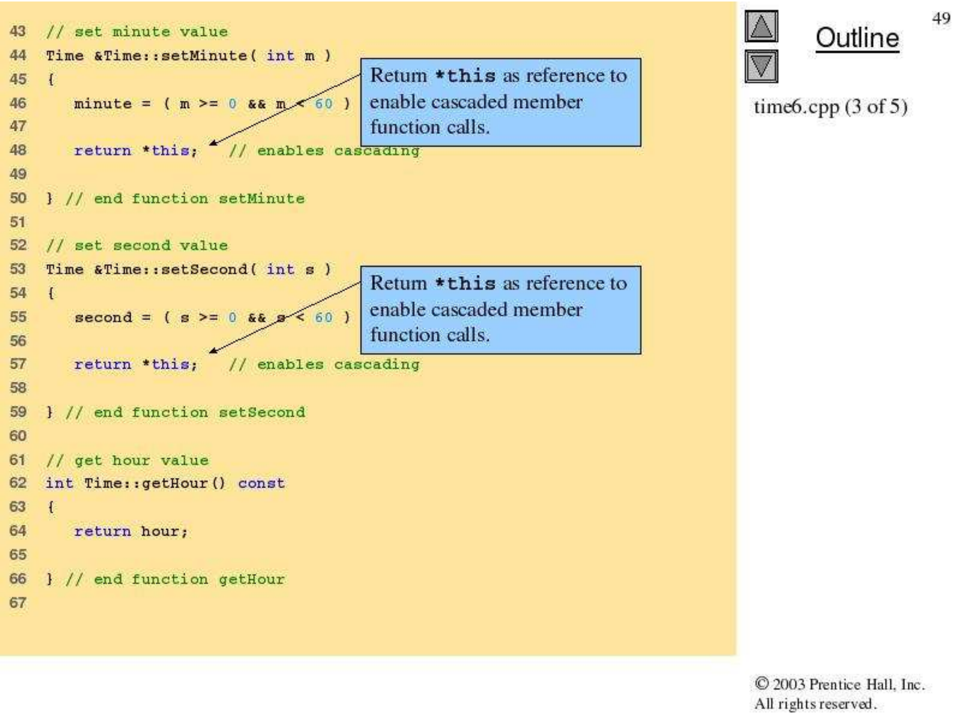
```
// set values of hour, minute, and second
Time &Time::setTime( int h, int m, int s )
{
   setHour( h );
   setMinute( m );
   setSecond( s );

   return *this;   // enables cascading

} // end function setTime

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

   return *this;   // enables cascading

} // end function setHour

```

Return *this as reference to enable cascaded member function calls.

Return *this as reference to enable cascaded member function calls.

© 2003 Prentice Hall, Inc.
All rights reserved.

Figure 2.21: **Time** class member-function definitions modified to enable cascaded member-function calls. (part 1 of 3)



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

   return *this;   // enables cascading

} // end function setMinute

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

   return *this;   // enables cascading

} // end function setSecond

// get hour value
int Time::getHour() const
{
   return hour;

} // end function getHour

```

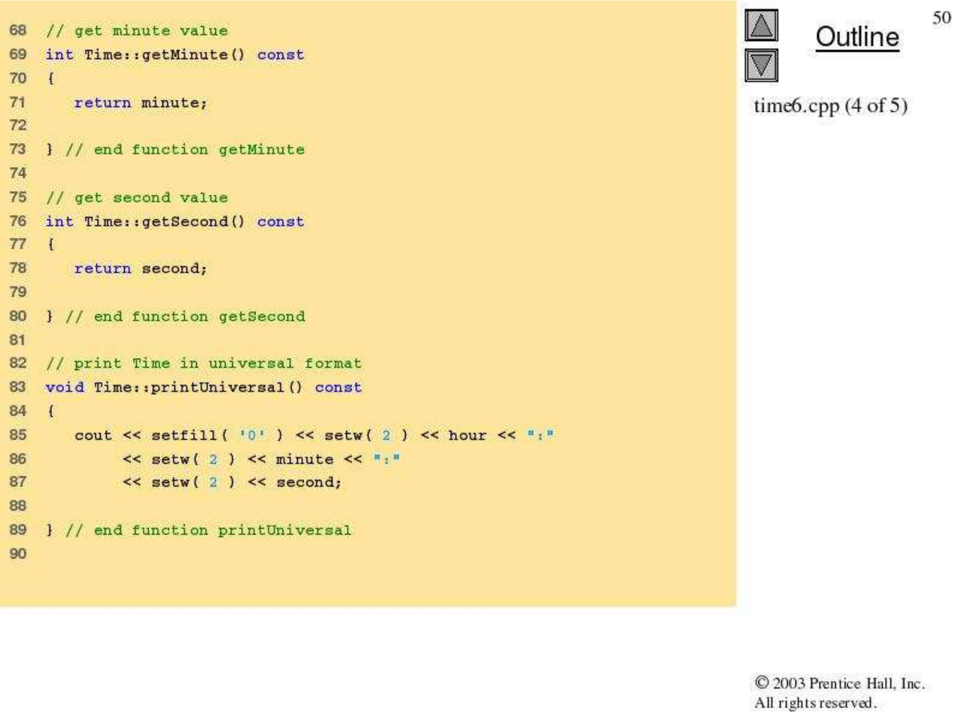
Return `*this` as reference to enable cascaded member function calls.

Return `*this` as reference to enable cascaded member function calls.

49

Outline

time6.cpp (3 of 5)



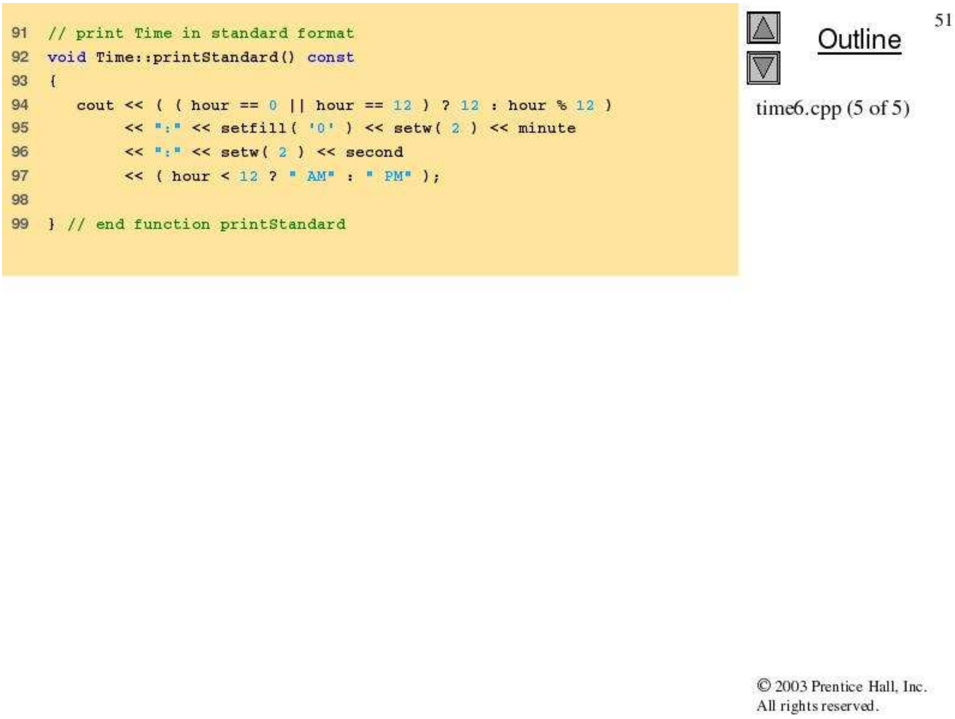
```
// get minute value
int Time::getMinute() const
{
   return minute;

} // end function getMinute

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

} // end function getSecond

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

} // end function printUniversal

```

50

Outline

time6.cpp (4 of 5)

Figure 2.22: **Time** class member-function definitions modified to enable cascaded member-function calls. (part 2 of 3)



```
// print Time in standard format
void Time::printStandard() const
{
   cout << ( ( hour == 0 || hour == 12 ) ? 12 : hour % 12 )
        << ":" << setfill( '0' ) << setw( 2 ) << minute
        << ":" << setw( 2 ) << second
        << ( hour < 12 ? " AM" : " PM" );

} // end function printStandard
```

51

Outline

time6.cpp (5 of 5)

Figure 2.23: **Time** class member-function definitions modified to enable cascaded member-function calls. (part 3 of 3)

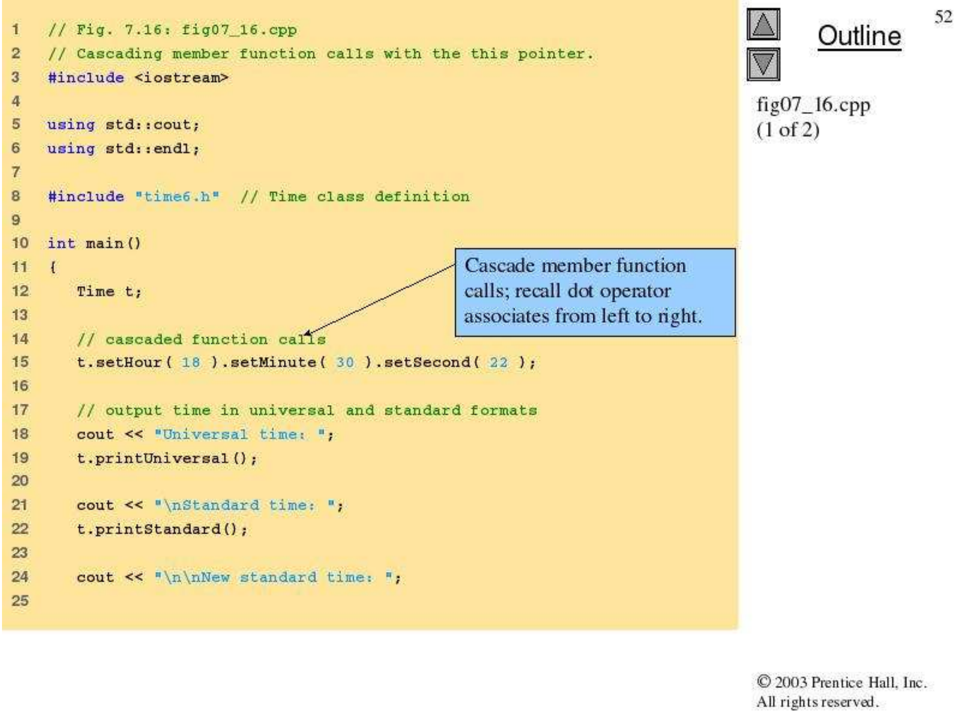
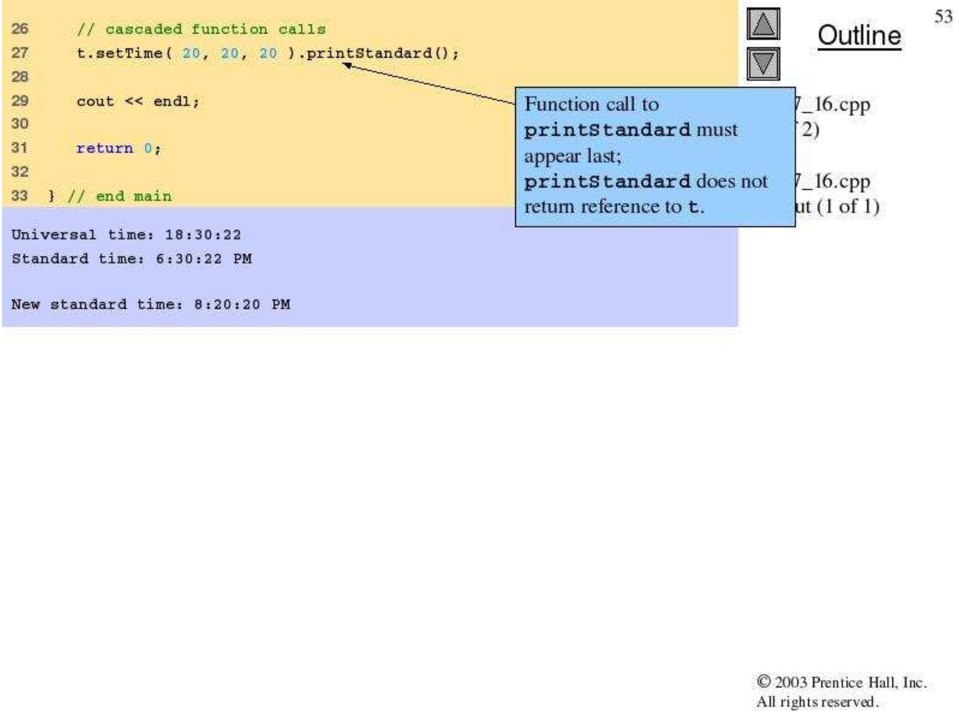


fig07_16.cpp
(1 of 2)

```
// Fig. 7.16: fig07_16.cpp
// Cascading member function calls with the this pointer.
#include <iostream>

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

#include "time6.h"  // Time class definition

int main()
{
   Time t;

   // cascaded function calls
   t.setHour( 18 ).setMinute( 30 ).setSecond( 22 );

   // output time in universal and standard formats
   cout << "Universal time: ";
   t.printUniversal();

   cout << "\nStandard time: ";
   t.printStandard();

   cout << "\n\nNew standard time: ";

```

Cascade member function calls; recall dot operator associates from left to right.

© 2003 Prentice Hall, Inc.
All rights reserved.



7_16.cpp
2)

7_16.cpp
ut (1 of 1)

```
   // cascaded function calls
   t.setTime( 20, 20, 20 ).printStandard();

   cout << endl;

   return 0;

} // end main
```

```
Universal time: 18:30:22
Standard time: 6:30:22 PM

New standard time: 8:20:20 PM
```

Function call to **printStandard** must appear last; **printStandard** does not return reference to **t**.

© 2003 Prentice Hall, Inc.
All rights reserved.

Figure 2.24: Cascading member-function calls.

2.5 Dynamic Memory Management with Operators new and delete

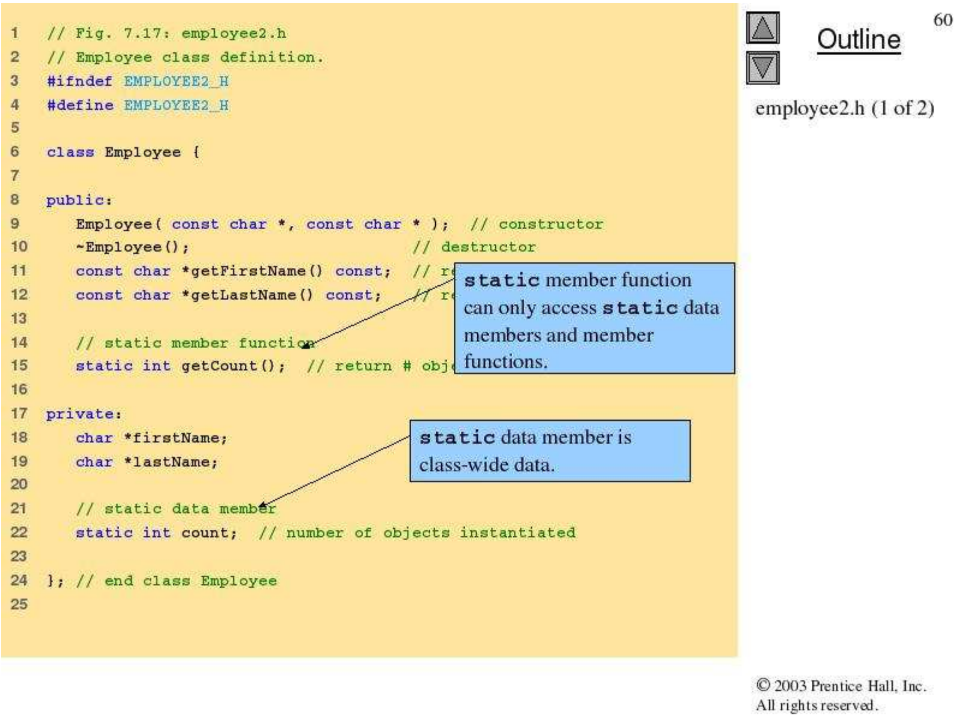
- Dynamic memory management
  - Control allocation and deallocation of memory
  - Operators **new** and **delete**
    - Include standard header <**new**>; Access to standard version of **new**
- **new**
  - Consider
    - **Time *timePtr;**
    - **timePtr = new Time;**
  - **new** operator
    - Creates object of proper size for type **Time**; Error if no space in memory for object
    - Calls default constructor for object
    - Returns pointer of specified type
  - Providing initializers
    - **double *ptr = new double(3.14159);**
    - **Time *timePtr = new Time(12, 0, 0);**
  - Allocating arrays; **int *gradesArray = new int[10];**
- **delete**
  - Destroy dynamically allocated object and free space
  - Consider; **delete timePtr;**
  - Operator **delete**
    - Calls destructor for object
    - Deallocates memory associated with object; Memory can be reused to allocate other objects
  - Deallocating arrays
    - **delete [] gradesArray;** ; Deallocates array to which **gradesArray** points
    - If pointer to array of objects
      - First calls destructor for each object in array
      - Then deallocates memory

2.6 static Class Members

Each object of a class has its own copy of all the **data members** of the class. in certain cases, only one copy of a variable should be shared by all objects of a class.

- **static** class variable
  - "Class-wide" data; Property of class, not specific object of class
  - Efficient when single copy of data is enough; Only the **static** variable has to be updated
  - May seem like global variables, but have class scope; Only accessible to objects of same class
  - Initialized exactly once at file scope
  - Exist even if no objects of class exist
  - Can be **public**, **private** or **protected**
- Accessing static class variables
  - Accessible through any object of class
  - **public static** variables
    - Can also be accessed using binary scope resolution operator(**::**)
    - **Employee::count**
- **private static** variables
  - When no class member objects exist
    - Can only be accessed via **public static** member function
    - To call **public static** member function combine class name, binary scope resolution operator (**::**) and function name; **Employee::getCount()**
- static member functions
  - Cannot access non-**static** data or functions
  - No **this** pointer for **static** functions; **static** data members and **static** member functions exist independent of objects

The programs of Figs. 2.25-2.28 demonstrates a **private static** data member called **count** and a **public static** member function called **getCount**. Figure 2.28 uses function **getCount** to determine the number of **Employee** objects currently instantiated.



60

Outline

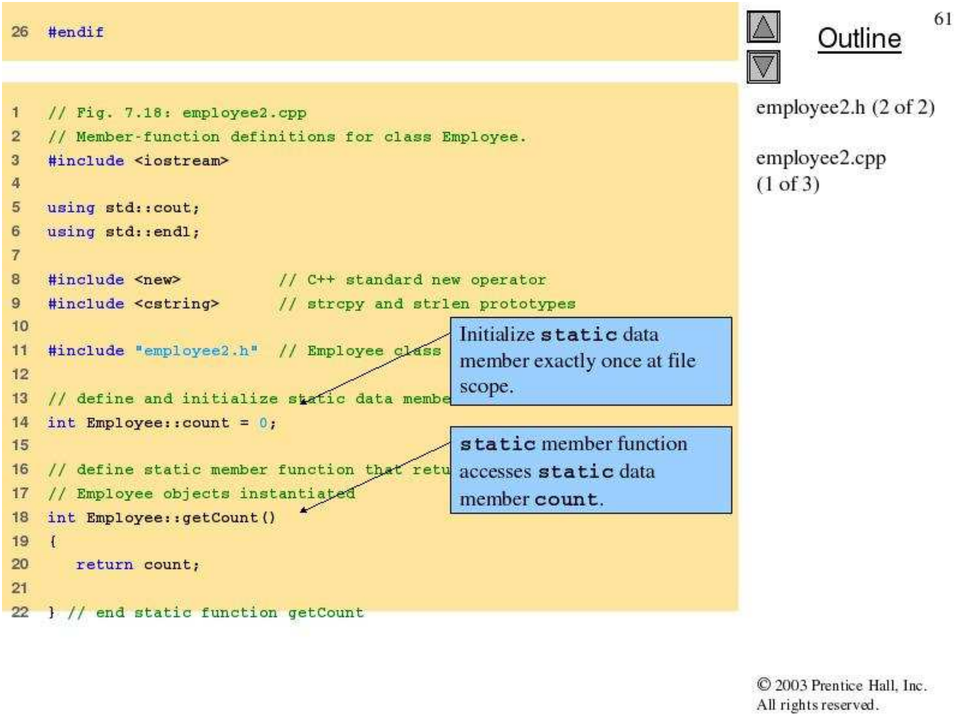
employee2.h (1 of 2)

```
// Fig. 7.17: employee2.h
// Employee class definition.
#ifndef EMPLOYEE2_H
#define EMPLOYEE2_H

class Employee {

public:
   Employee( const char *, const char * );  // constructor
   ~Employee();                            // destructor
   const char *getFirstName() const;  // r
   const char *getLastName() const;   // r

   // static member function
   static int getCount();  // return # obj

private:
   char *firstName;
   char *lastName;

   // static data member
   static int count;  // number of objects instantiated

}; // end class Employee

```

static member function can only access **static** data members and member functions.

static data member is class-wide data.

Figure 2.25: **Employee** class definition with a **static** data member to track the number **Employee** objects in memory.



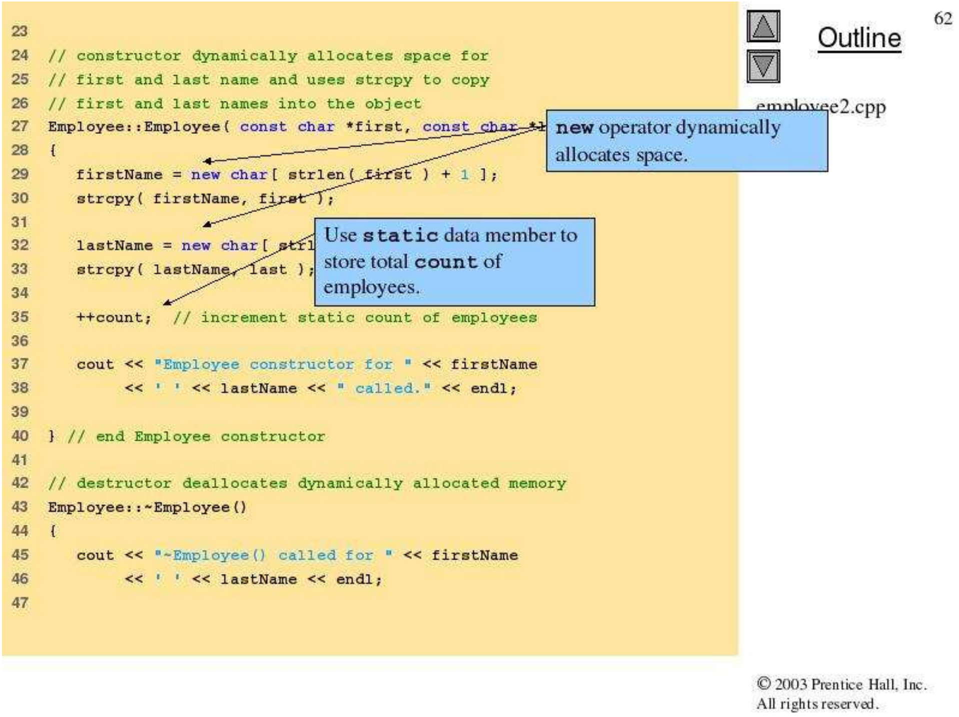
```
#endif
```

```
// Fig. 7.18: employee2.cpp
// Member-function definitions for class Employee.
#include <iostream>

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

#include <new>           // C++ standard new operator
#include <cstring>       // strcpy and strlen prototypes

#include "employee2.h"   // Employee class

// define and initialize static data member
int Employee::count = 0;

// define static member function that returns number of
// Employee objects instantiated
int Employee::getCount()
{
   return count;

} // end static function getCount
```

Initialize **static** data member exactly once at file scope.

static member function accesses **static** data member **count**.

© 2003 Prentice Hall, Inc.
All rights reserved.



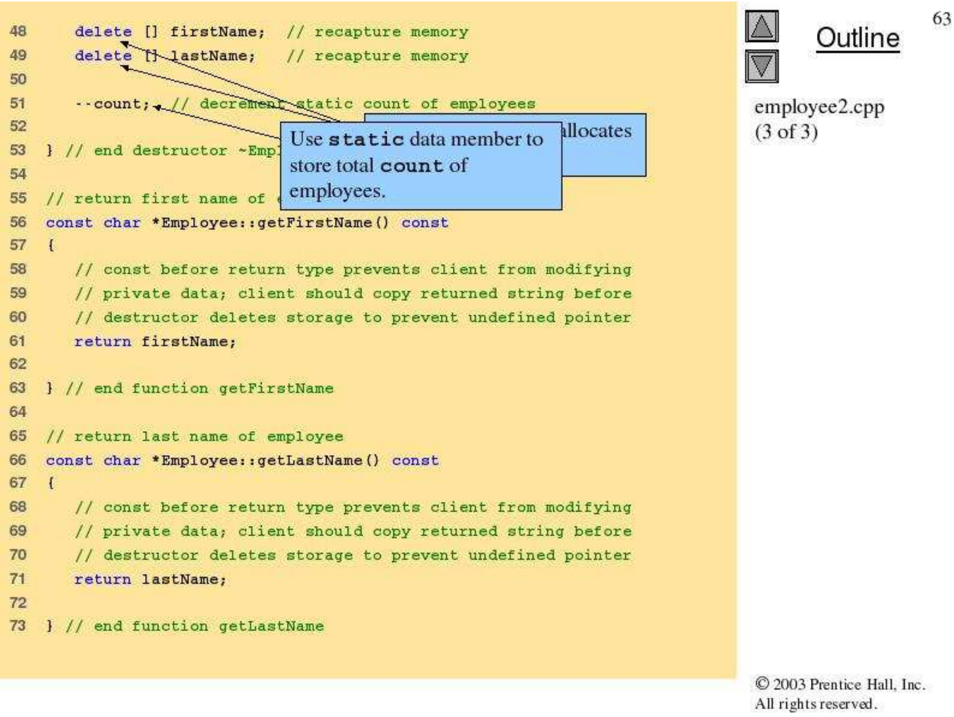
```

// constructor dynamically allocates space for
// first and last name and uses strcpy to copy
// first and last names into the object
Employee::Employee( const char *first, const char *last )
{
   firstName = new char[ strlen( first ) + 1 ];
   strcpy( firstName, first );

   lastName = new char[ strlen( last ) + 1 ];
   strcpy( lastName, last );

   ++count;  // increment static count of employees

   cout << "Employee constructor for " << firstName
        << ' ' << lastName << " called." << endl;

} // end Employee constructor

// destructor deallocates dynamically allocated memory
Employee::~Employee()
{
   cout << "~Employee() called for " << firstName
        << ' ' << lastName << endl;

```

new operator dynamically allocates space.

Use **static** data member to store total **count** of employees.

© 2003 Prentice Hall, Inc.
All rights reserved.

Figure 2.26: **Employee** class member-function definitions. (part 1 of 2)



```
   delete [] firstName;  // recapture memory
   delete [] lastName;   // recapture memory

   --count;  // decrement static count of employees

} // end destructor ~Emp

// return first name of e
const char *Employee::getFirstName() const
{
   // const before return type prevents client from modifying
   // private data; client should copy returned string before
   // destructor deletes storage to prevent undefined pointer
   return firstName;

} // end function getFirstName

// return last name of employee
const char *Employee::getLastName() const
{
   // const before return type prevents client from modifying
   // private data; client should copy returned string before
   // destructor deletes storage to prevent undefined pointer
   return lastName;

} // end function getLastName
```

llocates

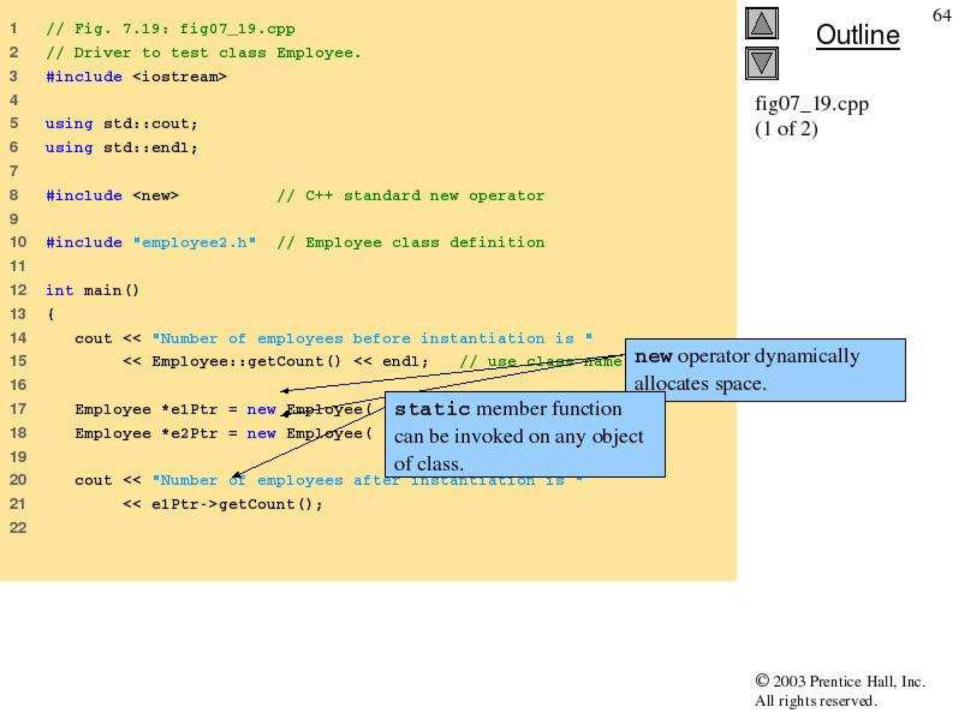
Use **static** data member to store total **count** of employees.

63

Outline

employee2.cpp
(3 of 3)

© 2003 Prentice Hall, Inc.
All rights reserved.



```
// Fig. 7.19: fig07_19.cpp
// Driver to test class Employee.
#include <iostream>

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

#include <new>          // C++ standard new operator

#include "employee2.h"  // Employee class definition

int main()
{
   cout << "Number of employees before instantiation is "
        << Employee::getCount() << endl;   // use class name

   Employee *e1Ptr = new Employee(
   Employee *e2Ptr = new Employee(

   cout << "Number of employees after instantiation is "
        << e1Ptr->getCount();

```

new operator dynamically allocates space.

static member function can be invoked on any object of class.

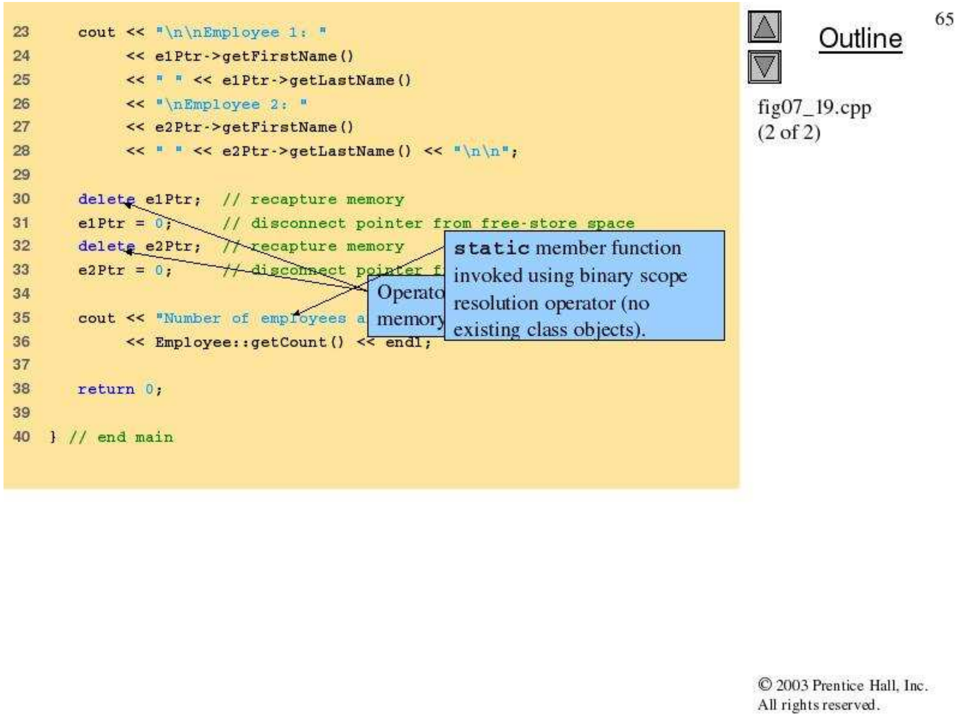
64

Outline

fig07_19.cpp
(1 of 2)

© 2003 Prentice Hall, Inc.
All rights reserved.

Figure 2.27: **Employee** class member-function definitions. (part 2 of 2) and **static** data member tracking the number of objects of a class. (part 1 of 2)

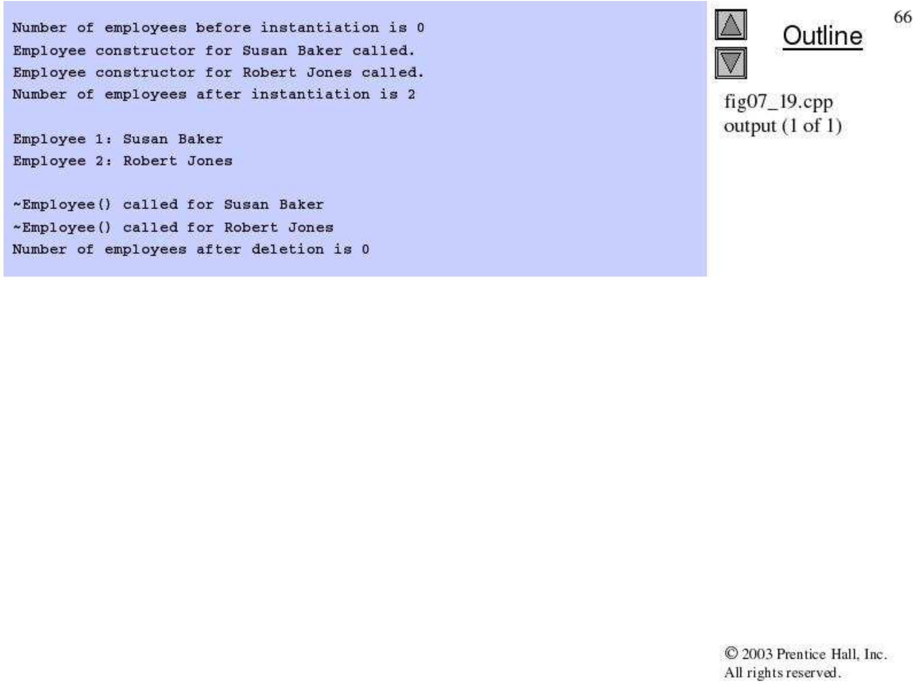


```
    cout << "\n\nEmployee 1: "
         << e1Ptr->getFirstName()
         << " " << e1Ptr->getLastName()
         << "\nEmployee 2: "
         << e2Ptr->getFirstName()
         << " " << e2Ptr->getLastName() << "\n\n";

    delete e1Ptr;  // recapture memory
    e1Ptr = 0;     // disconnect pointer from free-store space
    delete e2Ptr;  // recapture memory
    e2Ptr = 0;     // disconnect pointer f

    cout << "Number of employees a
         << Employee::getCount() << endl;

    return 0;

 } // end main
```

Operato
memory

static member function invoked using binary scope resolution operator (no existing class objects).

fig07_19.cpp
(2 of 2)



```
Number of employees before instantiation is 0
Employee constructor for Susan Baker called.
Employee constructor for Robert Jones called.
Number of employees after instantiation is 2

Employee 1: Susan Baker
Employee 2: Robert Jones

~Employee() called for Susan Baker
~Employee() called for Robert Jones
Number of employees after deletion is 0
```

fig07_19.cpp
output (1 of 1)

Figure 2.28: **static** data member tracking the number of objects of a class. (part 2 of 2)

2.7 Data Abstraction and Information Hiding

- **Information hiding**
  - Classes hide implementation details from clients
  - Example: stack data structure
    - Data elements added (pushed) onto top
    - Data elements removed (popped) from top
    - Last-in, first-out (LIFO) data structure
    - Client only wants LIFO data structure; Does not care how stack implemented
- **Data abstraction**; Describe functionality of class independent of implementation
- Abstract data types (ADTs)
  - Approximations/models of real-world concepts and behaviors; **int**, **float** are models for a numbers
  - Data representation
  - Operations allowed on those data
  - ADTs receive as much as attention today as structured programming did over the last two decades. (ADTs do not replace structured programming. rather, they provide an additional formalization that can further improve the program-development process.)
- C++ extensible; Standard data types cannot be changed, but new data types can be created

The job of high-level languages is to create a view convenient for programmers to use. There is no single accepted standard view-that is one reason why there are so many programming languages. Object-oriented programming in C++ presents yet another view.

The primary activity in C++ is creating new types (i.e., classes) and expressing the interactions among objects of those types.

2.7.1 Example: Array Abstract Data Type

An array is not much more than a pointer and some space in memory. Primitive capabilities! There are many operations that would be nice to perform with arrays, but there are not **built-in** C++. With C++ classes, the programmer can develop an array ADT is preferable to 'raw' arrays. Although the language is easy to extend with these new types, the base language itself is not changeable.

- ADT array
  - Subscript range checking
  - Arbitrary range of subscripts; Instead of having to start with 0
  - Array assignment
  - Array comparison
  - Array input/output
  - Arrays that know their sizes
  - Arrays that expand dynamically to accommodate more elements

2.7.2 Example: String Abstract Data Type

- Strings in C++
  - C++ does not provide built-in string data type; Maximizes performance
  - Provides mechanisms for creating and implementing string abstract data type; String ADT (Chapter 8)
  - ANSI/ISO standard **string** class (Chapter 19)

2.7.3 Example: Queue Abstract Data Type

A waiting line is also called a *queue*.

- Queue
  - FIFO; First in, first out
  - Enqueue; Put items in queue one at a time
  - Dequeue; Remove items from queue one at a time
- Queue ADT

  - Implementation hidden from clients; Clients may not manipulate data structure directly
  - Only queue member functions can access internal data
  - Queue ADT (Chapter 15)
  - Standard library **queue** class (Chapter 20)

The queue ADT guarantees the integrity of its internal data structure. Clients may not manipulate this data structure directly. Only the queue member functions have access to its internal data.

2.8 Container Classes and Iterators

- Container classes (collection classes)
  - Designed to hold collections of objects
  - Common services; Insertion, deletion, searching, sorting, or testing an item
  - Examples; Arrays, stacks, queues, trees and linked lists
- Iterator objects (iterators)
  - Returns next item of collection; Or performs some action on next item
  - Can have several iterators per container; Book with multiple bookmarks
  - Each iterator maintains own ”position”
  - Discussed further in Chapter 20

2.9 Proxy Classes

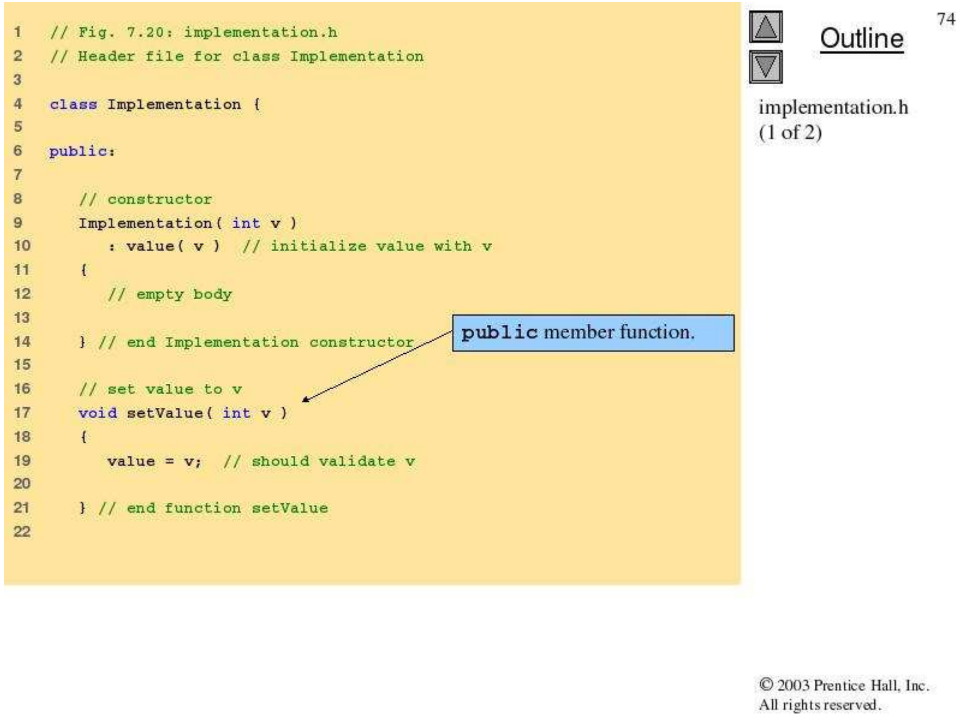
Sometimes, it is desirable to hide the implementation details of a class to prevent access to proprietary information (including private data) and proprietary program login in a class. Providing clients of your class with a **proxy class** that knows only the public interface to your class enables the clients to use your class's services without giving the client access to your class's implementation details.

- Proxy class

  - Hide implementation details of another class
  - Knows only **public** interface of class being hidden
  - Enables clients to use class's services without giving access to class's implementation

- Forward class declaration
  - Used when class definition only uses pointer to another class
  - Prevents need for including header file
  - Declares class before referencing
  - Format: **class ClassToLoad;**

Implementation of a proxy class is demonstrated in Figs. 2.29-2.31.



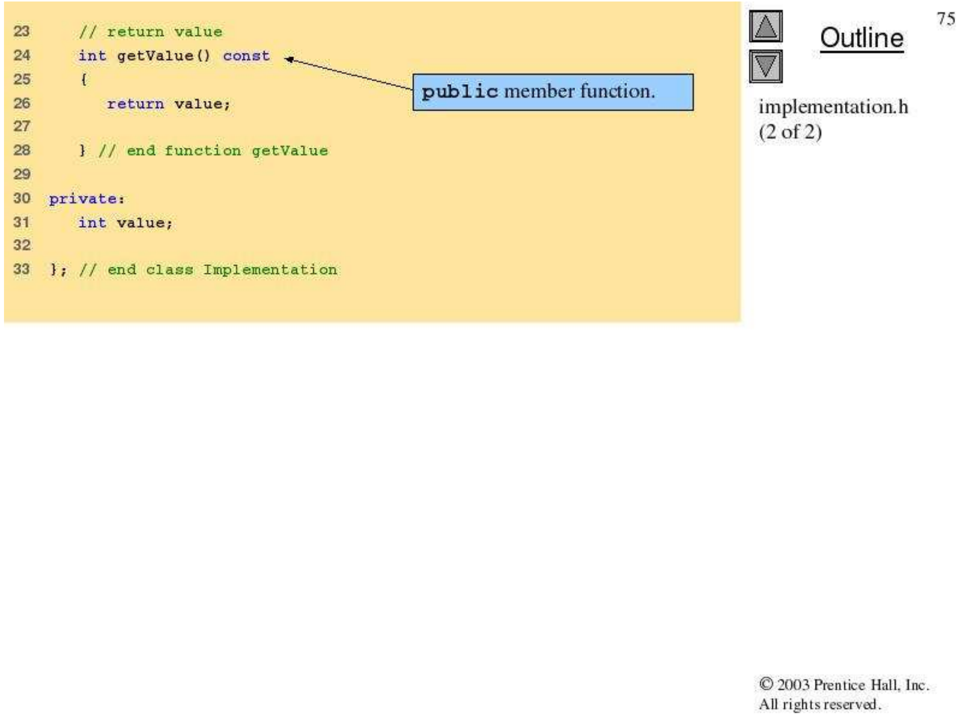
74

Outline

implementation.h
(1 of 2)

```
// Fig. 7.20: implementation.h
// Header file for class Implementation

class Implementation {

public:

   // constructor
   Implementation( int v )
      : value( v )  // initialize value with v
   {
      // empty body

   } // end Implementation constructor

   // set value to v
   void setValue( int v )
   {
      value = v;  // should validate v

   } // end function setValue

```

public member function.



75

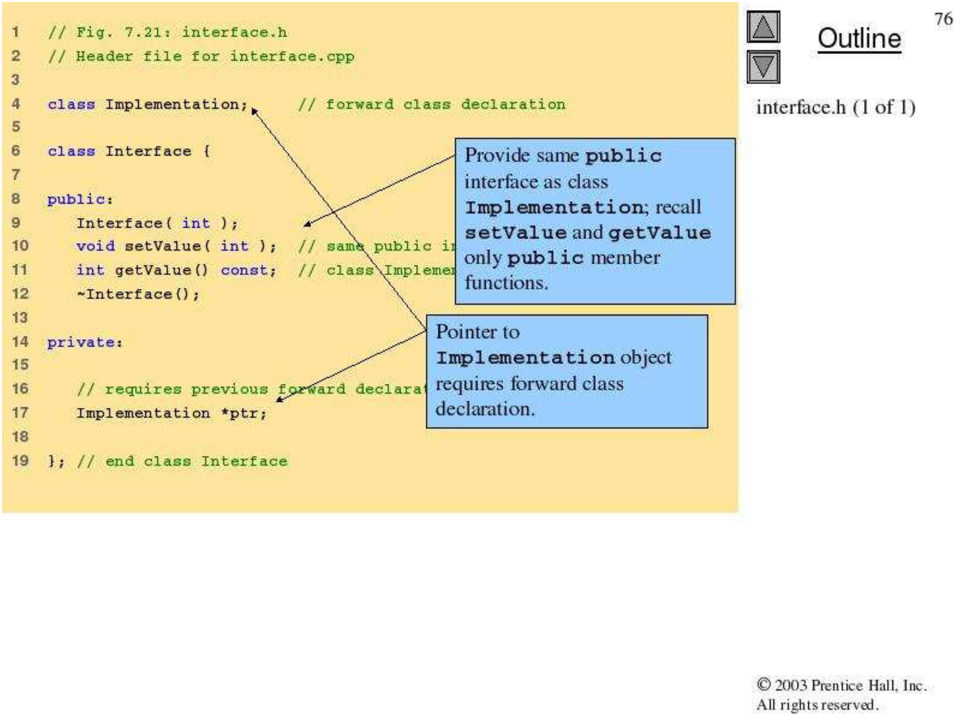
Outline

implementation.h
(2 of 2)

```
   // return value
   int getValue() const
   {
      return value;

   } // end function getValue

private:
   int value;

}; // end class Implementation
```

public member function.

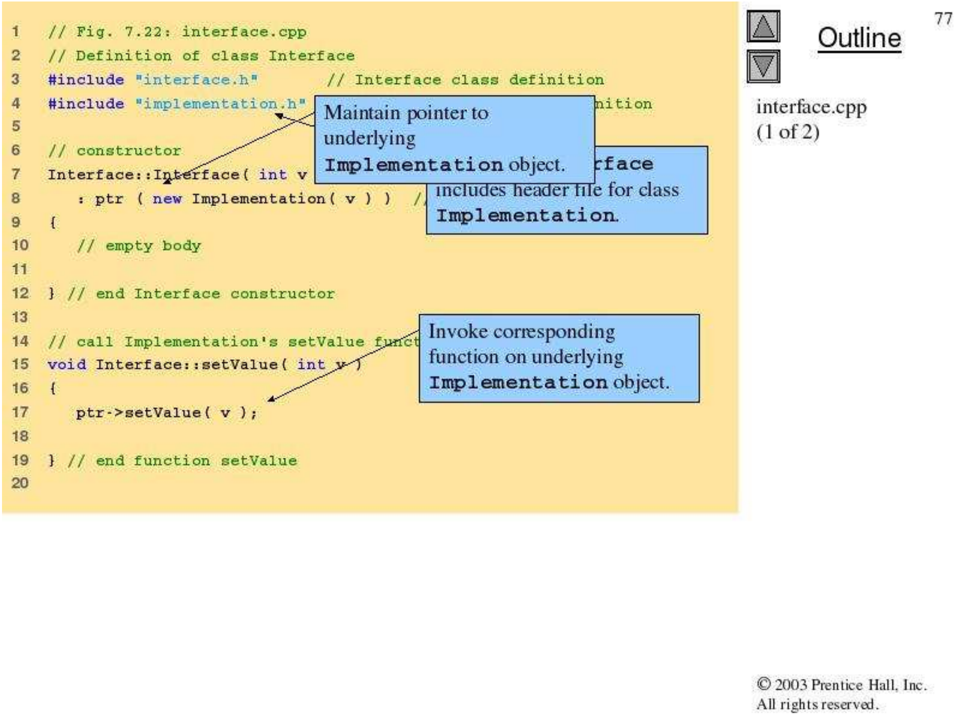
Figure 2.29: **Implementation** class definition.



```
// Fig. 7.21: interface.h
// Header file for interface.cpp

class Implementation;     // forward class declaration

class Interface {

public:
   Interface( int );
   void setValue( int );  // same public i
   int getValue() const;  // class Implemen
   ~Interface();

private:

   // requires previous forward declarat
   Implementation *ptr;

}; // end class Interface
```

Provide same **public** interface as class **Implementation**; recall **setValue** and **getValue** only **public** member functions.

Pointer to **Implementation** object requires forward class declaration.



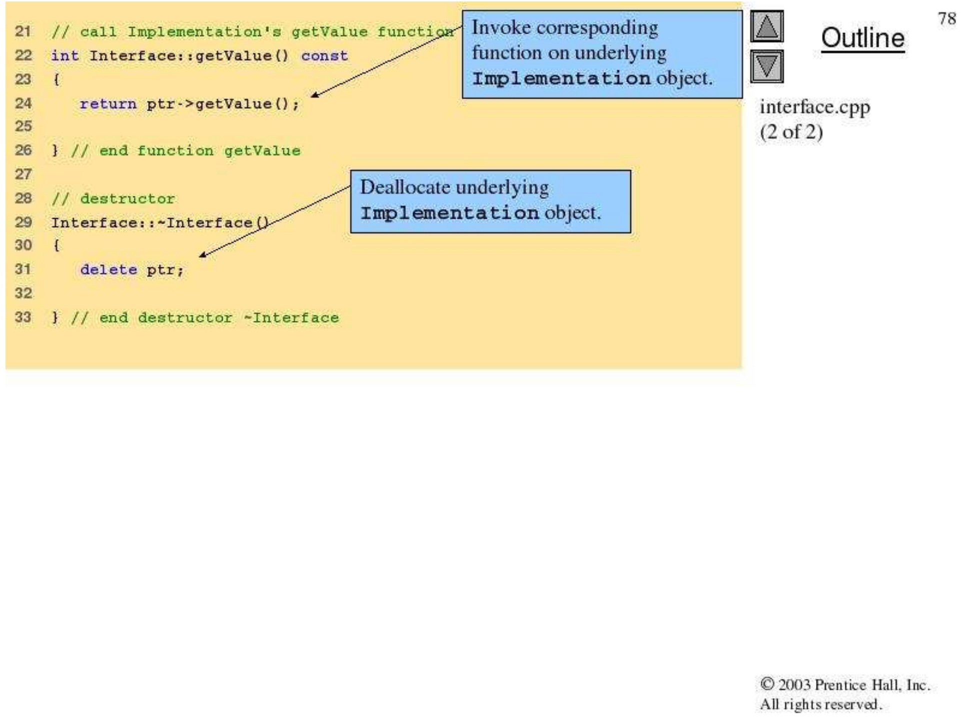
```
// Fig. 7.22: interface.cpp
// Definition of class Interface
#include "interface.h"       // Interface class definition
#include "implementation.h"                           nition

// constructor
Interface::Interface( int v
   : ptr ( new Implementation( v ) )   //
{
   // empty body

} // end Interface constructor

// call Implementation's setValue funct
void Interface::setValue( int v )
{
   ptr->setValue( v );

} // end function setValue

```

Maintain pointer to underlying **Implementation** object.

rface includes header file for class **Implementation**.

Invoke corresponding function on underlying **Implementation** object.

Figure 2.30: **Interface** class definition.



78

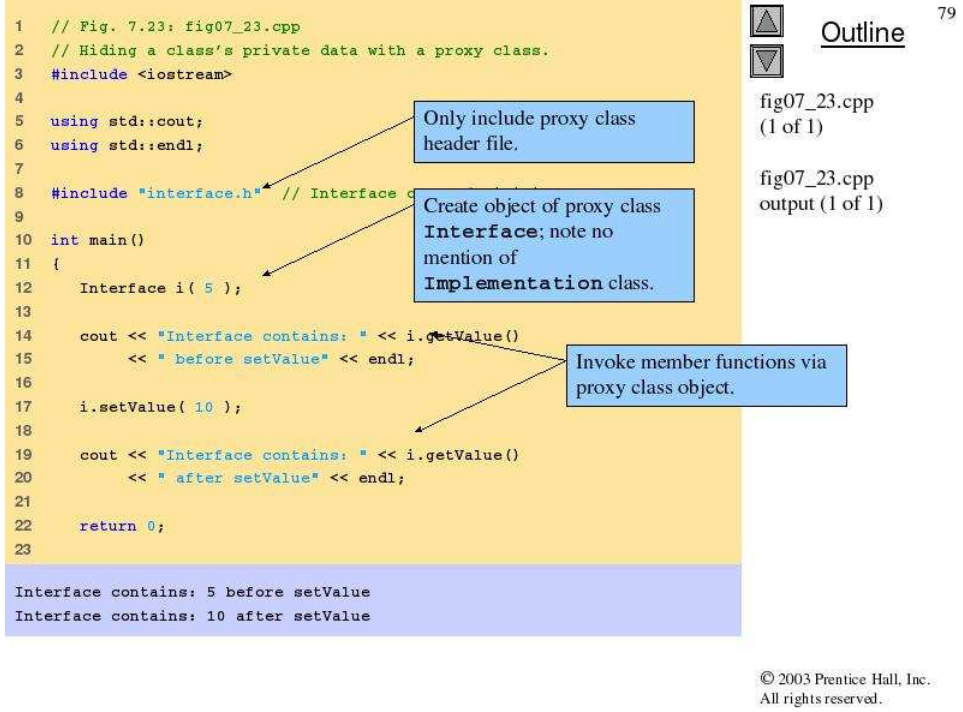
Outline

interface.cpp
(2 of 2)

```
// call Implementation's getValue function
int Interface::getValue() const
{
   return ptr->getValue();

} // end function getValue

// destructor
Interface::~Interface()
{
   delete ptr;

} // end destructor ~Interface
```

Invoke corresponding function on underlying **`Implementation`** object.

Deallocate underlying **`Implementation`** object.



79

Outline

fig07_23.cpp
(1 of 1)

fig07_23.cpp
output (1 of 1)

```
// Fig. 7.23: fig07_23.cpp
// Hiding a class's private data with a proxy class.
#include <iostream>

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

#include "interface.h"  // Interface c

int main()
{
   Interface i( 5 );

   cout << "Interface contains: " << i.getValue()
        << " before setValue" << endl;

   i.setValue( 10 );

   cout << "Interface contains: " << i.getValue()
        << " after setValue" << endl;

   return 0;

```

```
Interface contains: 5 before setValue
Interface contains: 10 after setValue
```

Only include proxy class header file.

Create object of proxy class **`Interface`**; note no mention of **`Implementation`** class.

Invoke member functions via proxy class object.

Figure 2.31: **Interface** class member-function definitions and Implementing a proxy class.

Chapter 3

Operator Overloading

3.1 Introduction

Manipulations on objects were accomplished by sending messages (in the form of member-function calls) to the object.

- Use operators with objects (operator overloading)
  - Clearer than function calls for certain classes
  - Operator sensitive to context
- Examples
  - <<; Stream insertion, bitwise left-shift
  - +; Performs arithmetic on multiple types (integers, floats, etc.)

3.2 Fundamentals of Operator Overloading

C++ programming is a type-sensitive and type-focused process. Operators provide programmers with a concise notation for expressing manipulations of objects of built-in types.

- Types
  - Built in (**int**, **char**) or user-defined
  - Can use existing operators with user-defined types; Cannot create new operators
- Overloading operators

  - Create a function for the class
  - Name function **operator** followed by symbol; **Operator+** for the addition operator +

- Using operators on a class object
  - It must be overloaded for that class
    - Exceptions:
    - Assignment operator, =; Memberwise assignment between objects
    - Address operator, &; Returns address of object
    - Both can be overloaded

- Overloading provides concise notation
  - **object2 = object1.add(object2);**
  - **object2 = object2 + object1;**

Overloading is especially appropriate for mathematical classes. These often require that a substantial set of operators be overloaded to ensure consistency with the way these mathematical classes are handled in the real world. Operator overloading is not automatic, however; the programmer must write operator-overloading functions to perform the desired operations. Sometimes these functions are best made member functions; sometimes they are best as **friend** functions; occasionally the can be made non-member, non-**friend** functions.

3.3 Restrictions on Operator Overloading

Most of C++'s operators can be overloaded.

- Cannot change
  - How operators act on built-in data types; i.e., cannot change integer addition
  - Precedence of operator (order of evaluation); Use parentheses to force order-of-operations
  - Associativity (left-to-right or right-to-left)
  - Number of operands; **&** is unitary, only acts on one operand

- Cannot create new operators
- Operators must be overloaded explicitly; Overloading + does not overload +=

3.4 Operator Functions As Class Members Vs. As Friend Functions

- Operator functions
  - Member functions
    - Use **this** keyword to implicitly get argument
    - Gets left operand for binary operators (like +)
    - Leftmost object must be of same class as operator
  - Non member functions
    - Need parameters for both operands
    - Can have object of different class than operator
    - Must be a **friend** to access **private** or **protected** data
  - Called when
    - Left operand of binary operator of same class
    - Single operand of unitary operator of same class
- Overloaded << operator
  - Left operand of type **ostream &**; Such as **cout** object in **cout << classObject**
  - Similarly, overloaded >> needs **istream &**
  - Thus, both must be non-member functions
- Commutative operators
  - May want + to be commutative; So both "**a + b**" and "**b + a**" work
  - Suppose we have two different classes
  - Overloaded operator can only be member function when its class is on left
    - **HugeIntClass + Long int**

    - ∗ Can be member function
  - When other way, need a non-member overload function; **Long int + HugeIntClass**

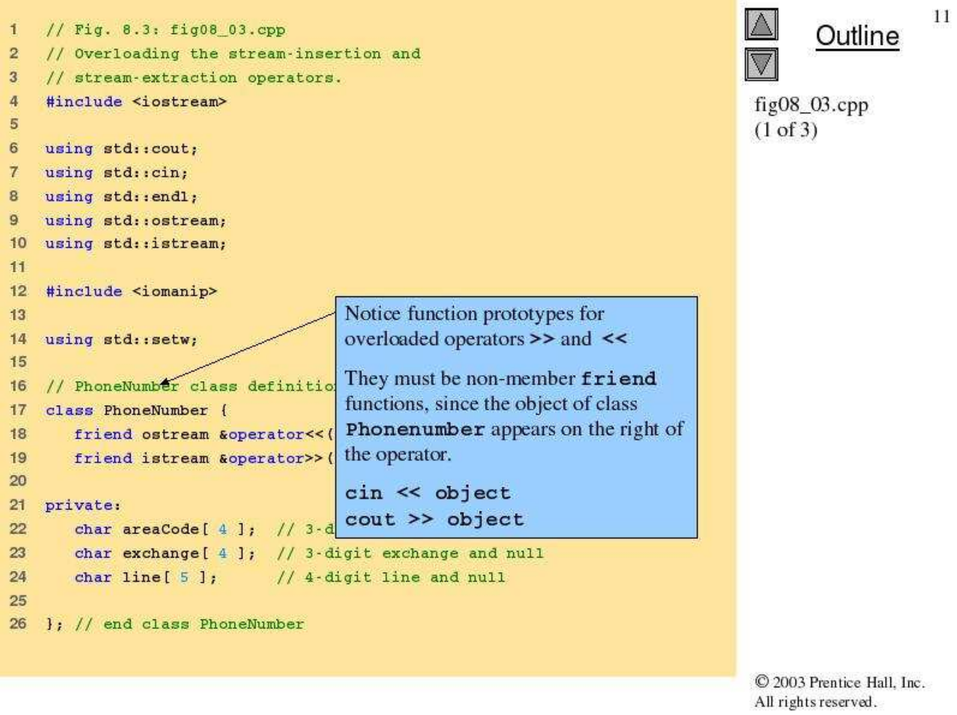
3.5 Overloading Stream-Insertion and Stream-Extraction Operators

- << and >>
  - Already overloaded to process each built-in type
  - Can also process a user-defined class
- Example program
  - Class **PhoneNumber**; Holds a telephone number
  - Print out formatted number automatically; **(123) 456-7890**

The program of Figs. 3.1-3.2 demonstrates overloading the stream-extraction and stream-insertion operators to handle data of a user-defined telephone number class called **PhoneNumber**.

3.6 Overloading Unary Operators

- Overloading unary operators
  - Non-**static** member function, no arguments
  - Non-member function, one argument; Argument must be class object or reference to class object
  - Remember, **static** functions only access **static** data



```
// Fig. 8.3: fig08_03.cpp
// Overloading the stream-insertion and
// stream-extraction operators.
#include <iostream>

using std::cout;
using std::cin;
using std::endl;
using std::ostream;
using std::istream;

#include <iomanip>

using std::setw;

// PhoneNumber class definitio
class PhoneNumber {
   friend ostream &operator<<(
   friend istream &operator>>(

private:
   char areaCode[ 4 ];  // 3-d
   char exchange[ 4 ];  // 3-digit exchange and null
   char line[ 5 ];      // 4-digit line and null

}; // end class PhoneNumber
```

Notice function prototypes for overloaded operators **>>** and **<<**

They must be non-member **friend** functions, since the object of class **Phonenumber** appears on the right of the operator.

cin << object
cout >> object

fig08_03.cpp
(1 of 3)

Figure 3.1: Overloaded stream-insertion and stream extraction operators. (part 1 of 2)

- Upcoming example (8.10)
  - Overload **!** to test for empty string
  - If non-**static** member function, needs no arguments
    - **!s** becomes **s.operator!()**
    - **class String { public: bool operator!() const; . . . };**
- If non-member function, needs one argument
  - **s!** becomes **operator!(s)**
  - **class String { friend bool operator!(const String &) ... }**

3.7 Overloading Binary Operators

- Overloading binary operators

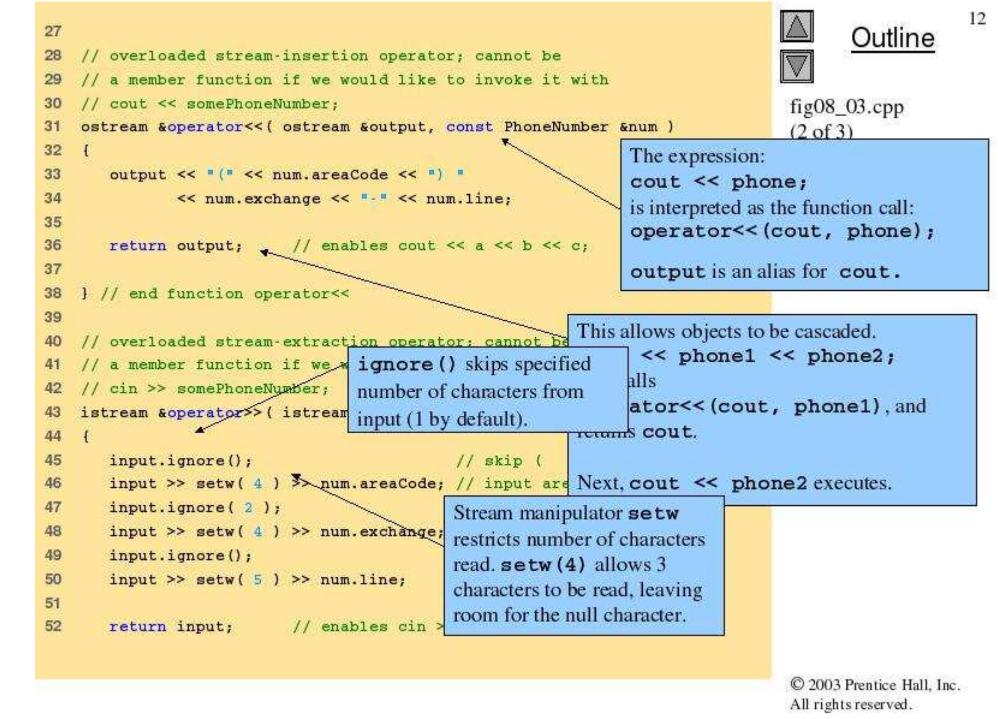


fig08_03.cpp
(2 of 3)

```
// overloaded stream-insertion operator; cannot be
// a member function if we would like to invoke it with
// cout << somePhoneNumber;
ostream &operator<<( ostream &output, const PhoneNumber &num )
{
   output << "(" << num.areaCode << ") "
          << num.exchange << "-" << num.line;

   return output;     // enables cout << a << b << c;

} // end function operator<<

// overloaded stream-extraction operator; cannot be
// a member function if we w
// cin >> somePhoneNumber;
istream &operator>>( istream
{
   input.ignore();                    // skip (
   input >> setw( 4 ) >> num.areaCode; // input are
   input.ignore( 2 );
   input >> setw( 4 ) >> num.exchange;
   input.ignore();
   input >> setw( 5 ) >> num.line;

   return input;      // enables cin >
```

The expression:
cout << phone;
is interpreted as the function call:
operator<<(cout, phone);

output is an alias for **cout.**

This allows objects to be cascaded.
<< phone1 << phone2;
alls
ator<<(cout, phone1), and
s **cout.**

Next, **cout << phone2** executes.

ignore() skips specified number of characters from input (1 by default).

Stream manipulator **setw** restricts number of characters read. **setw(4)** allows 3 characters to be read, leaving room for the null character.

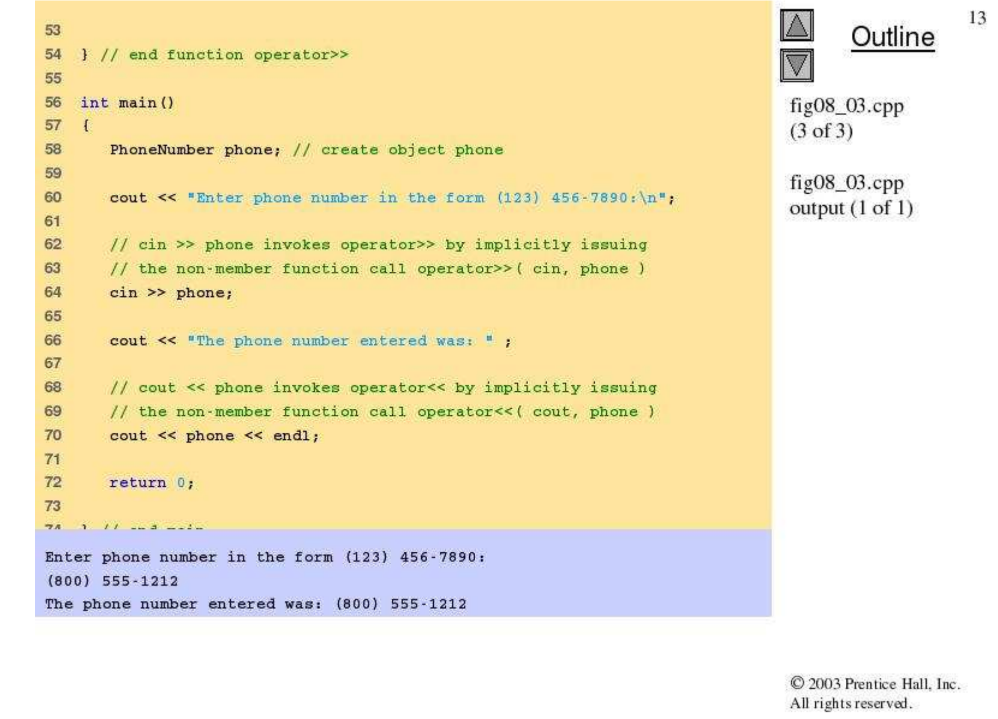


fig08_03.cpp
(3 of 3)

fig08_03.cpp
output (1 of 1)

```

} // end function operator>>

int main()
{
   PhoneNumber phone; // create object phone

   cout << "Enter phone number in the form (123) 456-7890:\n";

   // cin >> phone invokes operator>> by implicitly issuing
   // the non-member function call operator>>( cin, phone )
   cin >> phone;

   cout << "The phone number entered was: " ;

   // cout << phone invokes operator<< by implicitly issuing
   // the non-member function call operator<<( cout, phone )
   cout << phone << endl;

   return 0;

```

```
Enter phone number in the form (123) 456-7890:
(800) 555-1212
The phone number entered was: (800) 555-1212
```

Figure 3.2: Overloaded stream-insertion and stream extraction operators. (part 2 of 2)

  - Non-**static** member function, one argument
  - Non-member function, two arguments
  - One argument must be class object or reference

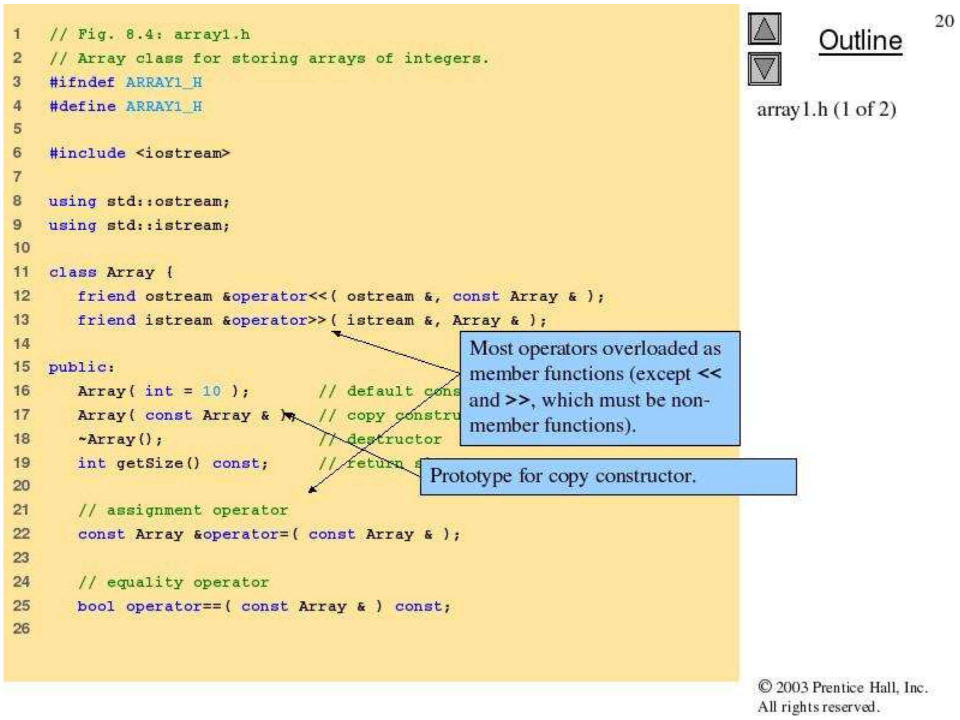
- Upcoming example
  - If non-**static** member function, needs one argument
    - **class String {**
    - **public:**
    - **const String &operator+=(const String &);**
    - **...**
    - **};**
  - **y += z** equivalent to **y.operator+=(z)**

3.8 Case Study: Array class

- Arrays in C++
  - No range checking
  - Cannot be compared meaningfully with ==
  - No array assignment (array names **const** pointers)
  - Cannot input/output entire arrays at once; One element at a time
- Example:Implement an Array class with
  - Range checking
  - Array assignment
  - Arrays that know their size
  - Outputting/inputting entire arrays with << and >>
  - Array comparisons with == and **!=**
- Copy constructor
  - Used whenever copy of object needed
    - Passing by value (return value or parameter)
    - Initializing an object with a copy of another; **Array newArray(oldArray);**

    - **newArray** copy of **oldArray**
  - Prototype for class **Array**
    - **Array(const Array &);**
    - *Must* take reference
      - Otherwise, pass by value
      - Tries to make copy by calling copy constructor ...
      - Infinite loop

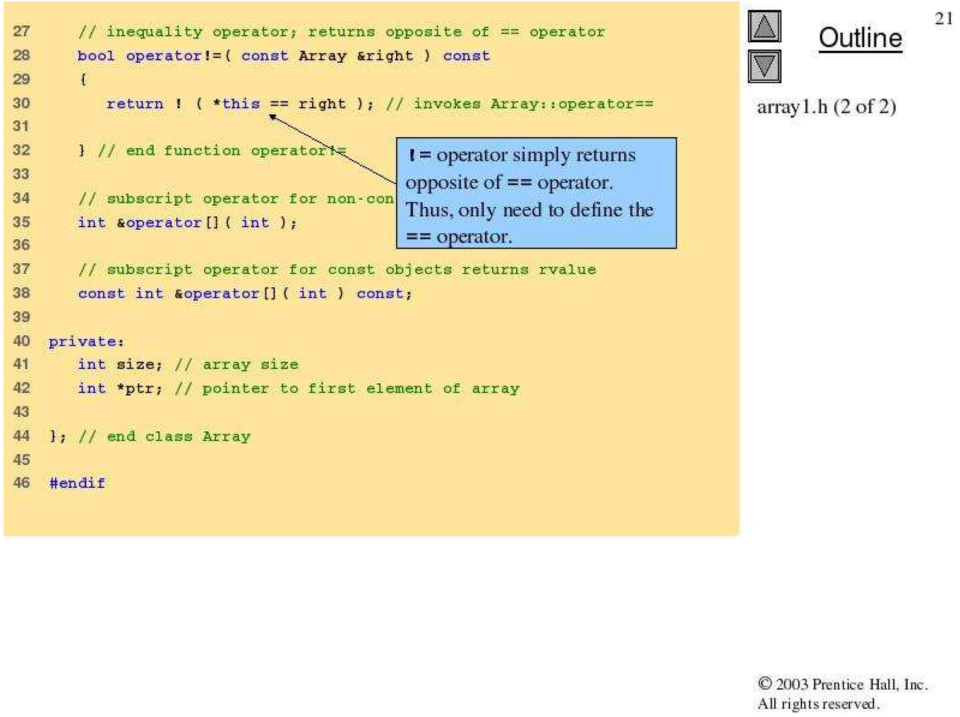
The program of Figs. 3.3-3.11 demonstrates class **Array** and its overloaded operators.



```
// Fig. 8.4: array1.h
// Array class for storing arrays of integers.
#ifndef ARRAY1_H
#define ARRAY1_H

#include <iostream>

using std::ostream;
using std::istream;

class Array {
   friend ostream &operator<<( ostream &, const Array & );
   friend istream &operator>>( istream &, Array & );

public:
   Array( int = 10 );         // default cons
   Array( const Array & );    // copy constru
   ~Array();                  // destructor
   int getSize() const;       // return s

   // assignment operator
   const Array &operator=( const Array & );

   // equality operator
   bool operator==( const Array & ) const;

```

Most operators overloaded as member functions (except << and >>, which must be non-member functions).

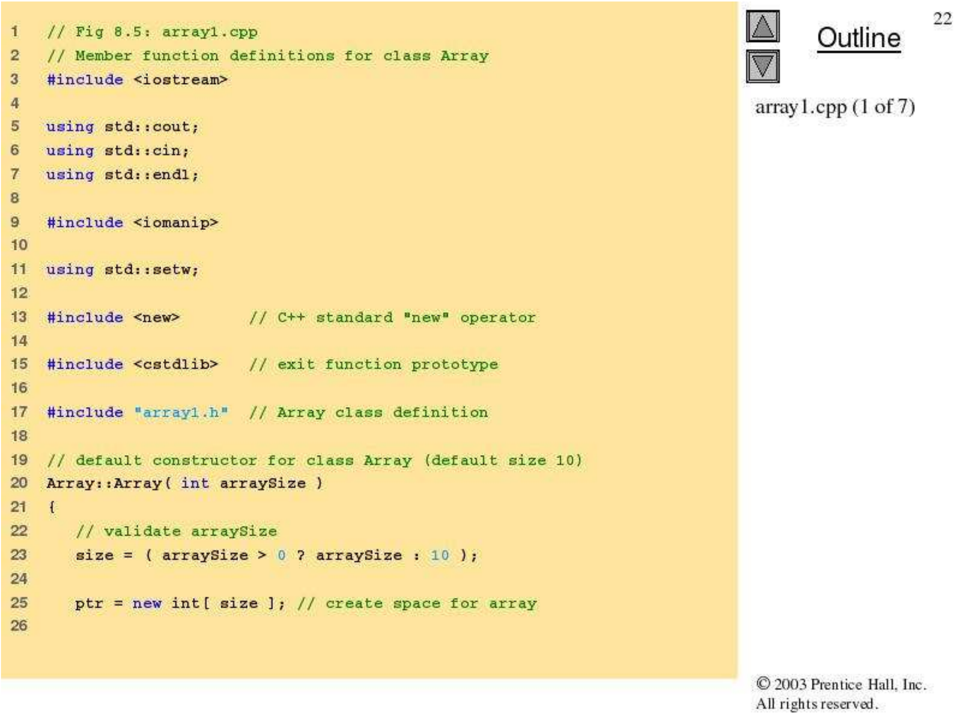
Prototype for copy constructor.



```
   // inequality operator; returns opposite of == operator
   bool operator!=( const Array &right ) const
   {
      return ! ( *this == right ); // invokes Array::operator==

   } // end function operator!=

   // subscript operator for non-con
   int &operator[]( int );

   // subscript operator for const objects returns rvalue
   const int &operator[]( int ) const;

private:
   int size; // array size
   int *ptr; // pointer to first element of array

}; // end class Array

#endif
```

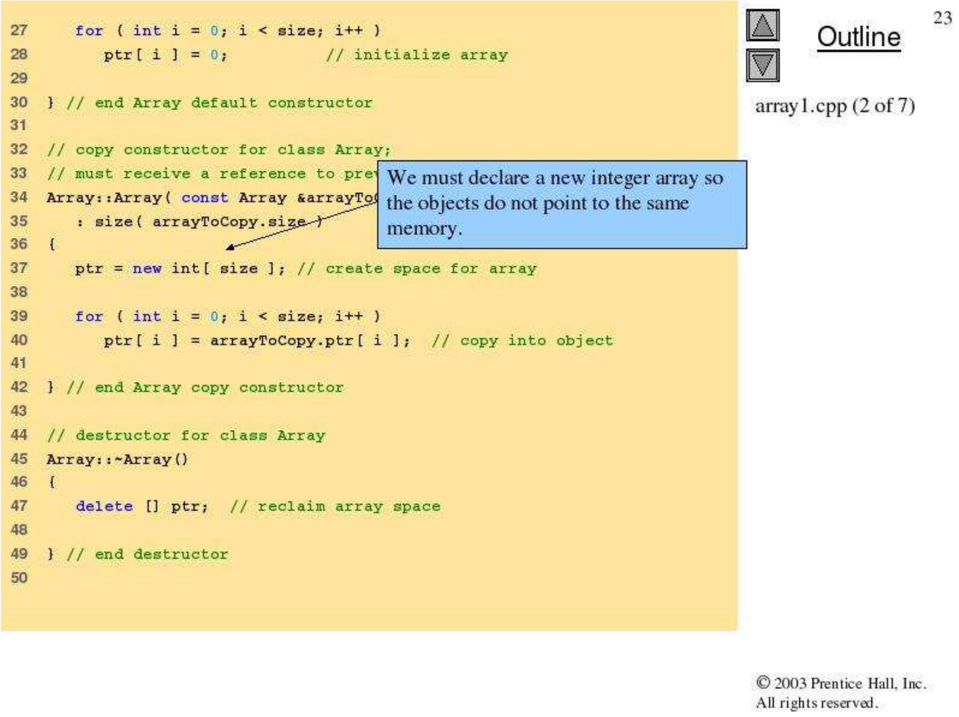
!= operator simply returns opposite of == operator. Thus, only need to define the == operator.

Figure 3.3: **Array** class definition with overloaded operators.



```
// Fig 8.5: array1.cpp
// Member function definitions for class Array
#include <iostream>

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

#include <iomanip>

using std::setw;

#include <new>       // C++ standard "new" operator

#include <cstdlib>   // exit function prototype

#include "array1.h"  // Array class definition

// default constructor for class Array (default size 10)
Array::Array( int arraySize )
{
   // validate arraySize
   size = ( arraySize > 0 ? arraySize : 10 );

   ptr = new int[ size ]; // create space for array

```

© 2003 Prentice Hall, Inc.
All rights reserved.

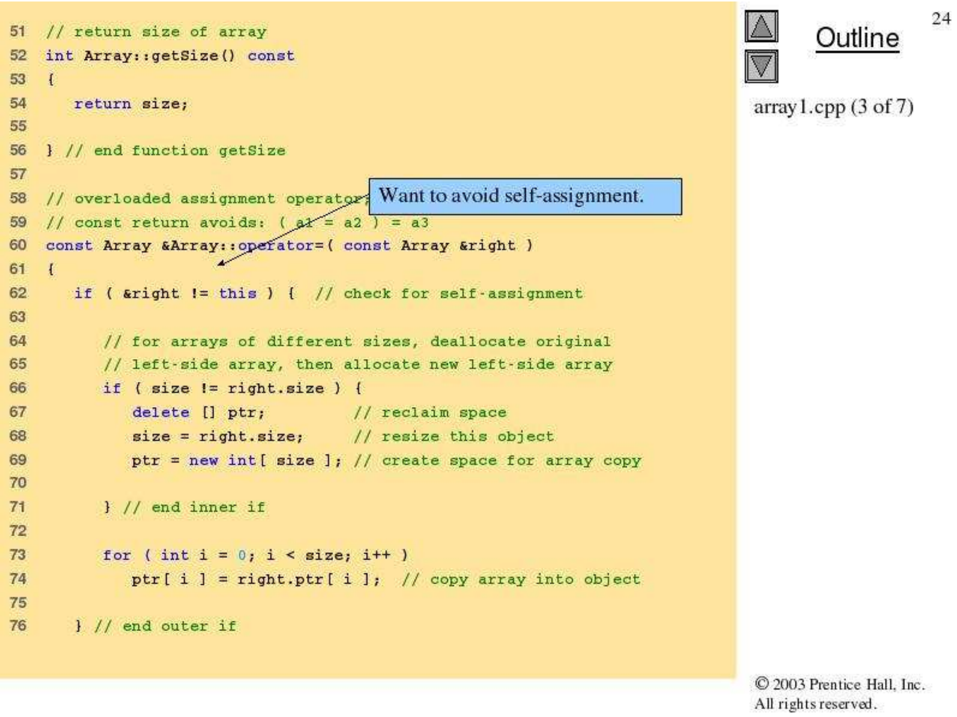


```
   for ( int i = 0; i < size; i++ )
      ptr[ i ] = 0;          // initialize array

} // end Array default constructor

// copy constructor for class Array;
// must receive a reference to prev
Array::Array( const Array &arrayToC
   : size( arrayToCopy.size )
{
   ptr = new int[ size ]; // create space for array

   for ( int i = 0; i < size; i++ )
      ptr[ i ] = arrayToCopy.ptr[ i ];  // copy into object

} // end Array copy constructor

// destructor for class Array
Array::~Array()
{
   delete [] ptr;  // reclaim array space

} // end destructor

```

We must declare a new integer array so the objects do not point to the same memory.

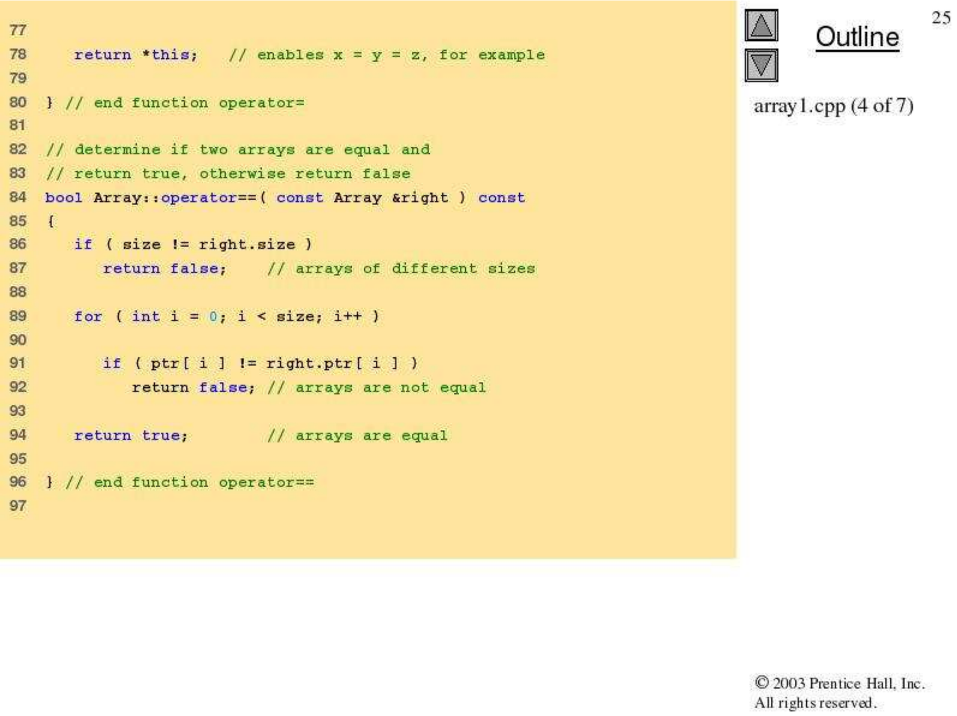
© 2003 Prentice Hall, Inc.
All rights reserved.

Figure 3.4: **Array** class member-and friend-function definitions. (part 1 of 4)



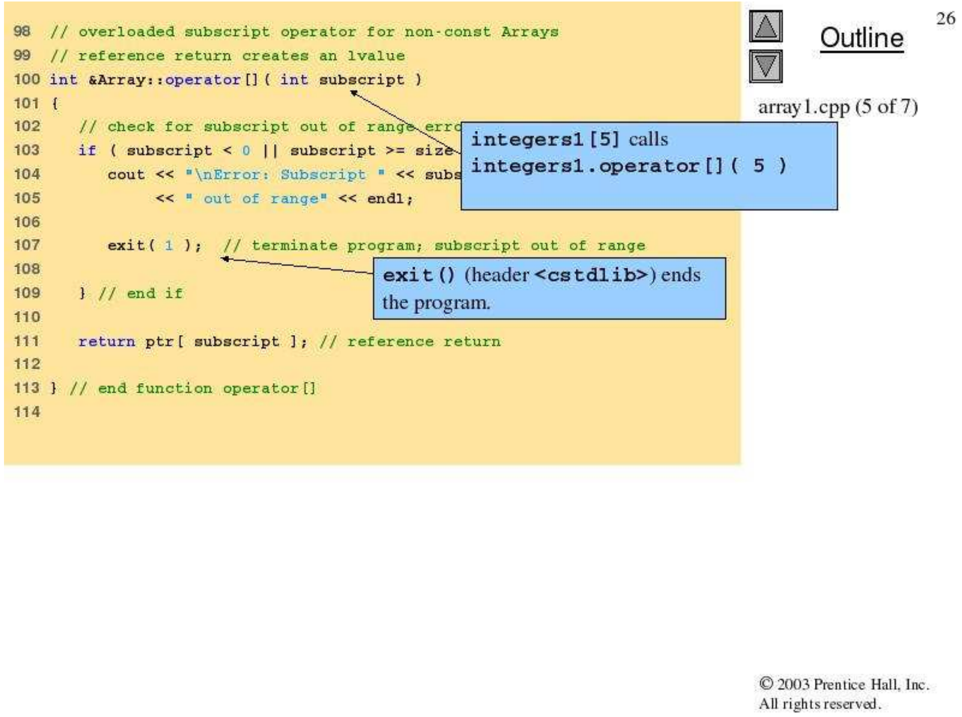
```
// return size of array
int Array::getSize() const
{
   return size;

} // end function getSize

// overloaded assignment operator
// const return avoids: ( a1 = a2 ) = a3
const Array &Array::operator=( const Array &right )
{
   if ( &right != this ) {  // check for self-assignment

      // for arrays of different sizes, deallocate original
      // left-side array, then allocate new left-side array
      if ( size != right.size ) {
         delete [] ptr;         // reclaim space
         size = right.size;     // resize this object
         ptr = new int[ size ]; // create space for array copy

      } // end inner if

      for ( int i = 0; i < size; i++ )
         ptr[ i ] = right.ptr[ i ];  // copy array into object

   } // end outer if
```

Want to avoid self-assignment.



```

   return *this;   // enables x = y = z, for example

} // end function operator=

// determine if two arrays are equal and
// return true, otherwise return false
bool Array::operator==( const Array &right ) const
{
   if ( size != right.size )
      return false;    // arrays of different sizes

   for ( int i = 0; i < size; i++ )

      if ( ptr[ i ] != right.ptr[ i ] )
         return false; // arrays are not equal

   return true;        // arrays are equal

} // end function operator==

```

Figure 3.5: **Array** class member-and friend-function definitions. (part 2 of 4)



26

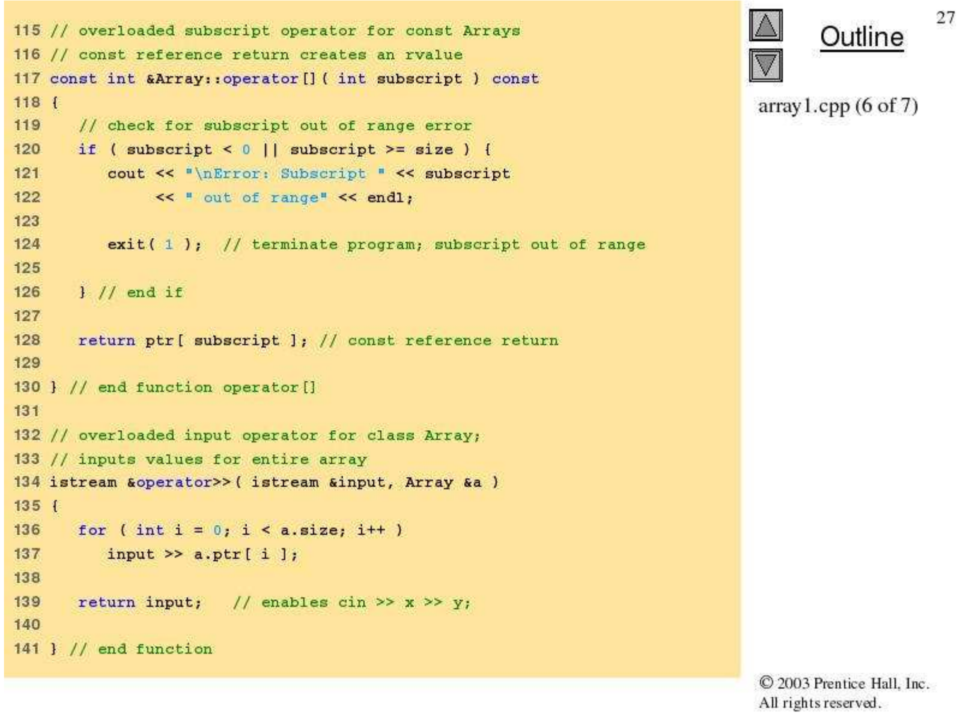
Outline

array1.cpp (5 of 7)

```
// overloaded subscript operator for non-const Arrays
// reference return creates an lvalue
int &Array::operator[]( int subscript )
{
   // check for subscript out of range erro
   if ( subscript < 0 || subscript >= size
      cout << "\nError: Subscript " << subs
           << " out of range" << endl;

      exit( 1 );  // terminate program; subscript out of range

   } // end if

   return ptr[ subscript ]; // reference return

} // end function operator[]

```

integers1[5] calls **integers1.operator[](5)**

exit() (header **<cstdlib>**) ends the program.



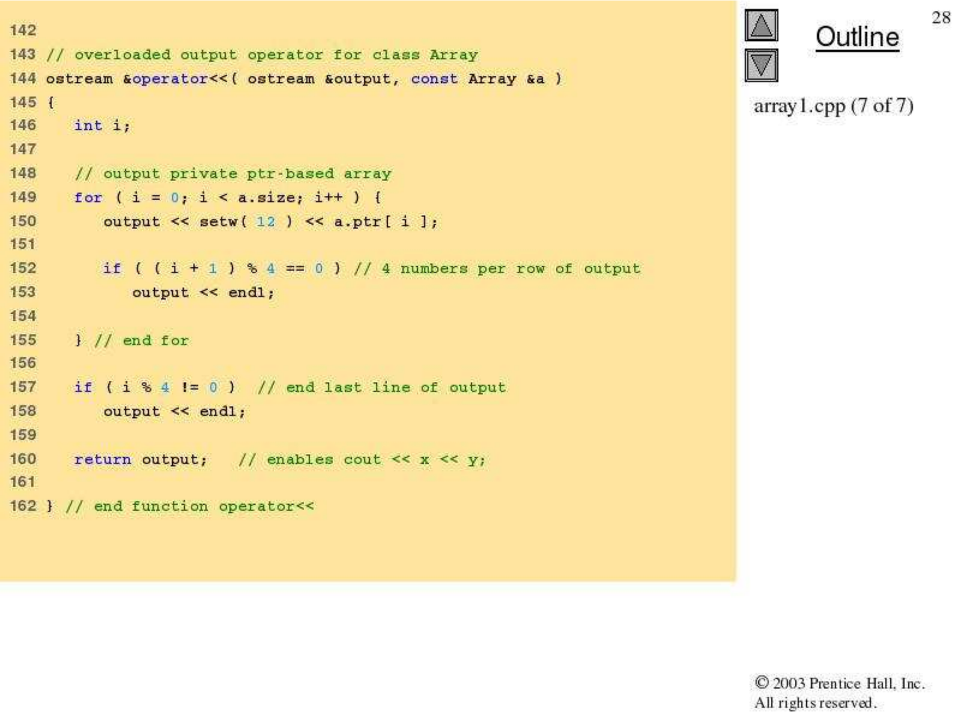
27

Outline

array1.cpp (6 of 7)

```
// overloaded subscript operator for const Arrays
// const reference return creates an rvalue
const int &Array::operator[]( int subscript ) const
{
   // check for subscript out of range error
   if ( subscript < 0 || subscript >= size ) {
      cout << "\nError: Subscript " << subscript
           << " out of range" << endl;

      exit( 1 );  // terminate program; subscript out of range

   } // end if

   return ptr[ subscript ]; // const reference return

} // end function operator[]

// overloaded input operator for class Array;
// inputs values for entire array
istream &operator>>( istream &input, Array &a )
{
   for ( int i = 0; i < a.size; i++ )
      input >> a.ptr[ i ];

   return input;   // enables cin >> x >> y;

} // end function
```

Figure 3.6: **Array** class member-and friend-function definitions. (part 3 of 4)



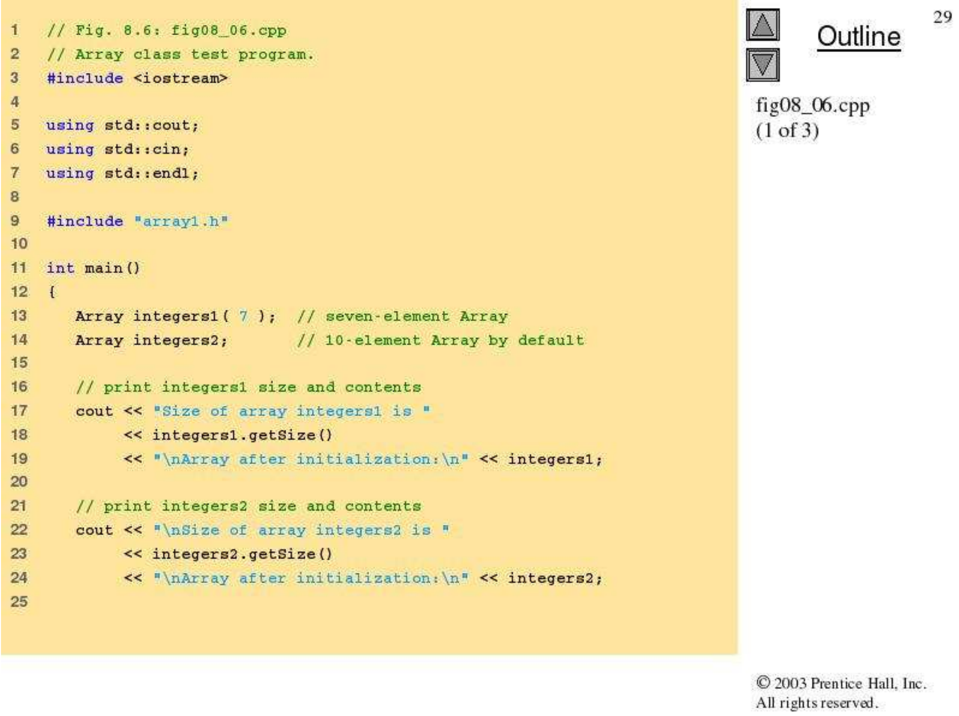
28

Outline

array1.cpp (7 of 7)

```

// overloaded output operator for class Array
ostream &operator<<( ostream &output, const Array &a )
{
   int i;

   // output private ptr-based array
   for ( i = 0; i < a.size; i++ ) {
      output << setw( 12 ) << a.ptr[ i ];

      if ( ( i + 1 ) % 4 == 0 ) // 4 numbers per row of output
         output << endl;

   } // end for

   if ( i % 4 != 0 )  // end last line of output
      output << endl;

   return output;   // enables cout << x << y;

} // end function operator<<
```

Figure 3.7: Overloaded stream-insertion and stream extraction operators. (part 4 of 2)

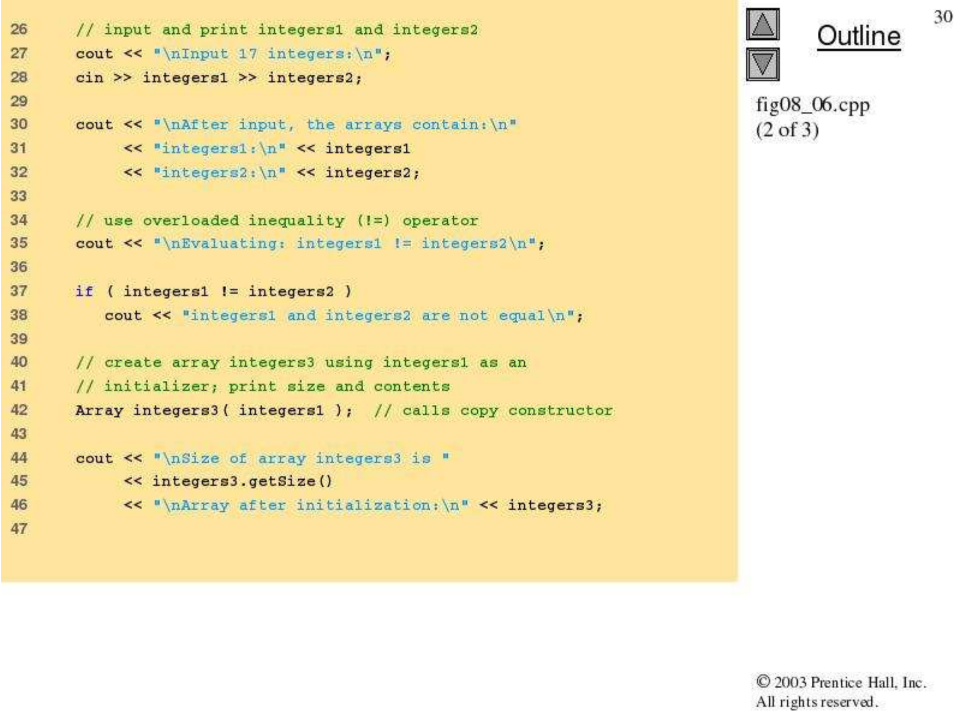


```
// Fig. 8.6: fig08_06.cpp
// Array class test program.
#include <iostream>

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

#include "array1.h"

int main()
{
   Array integers1( 7 );  // seven-element Array
   Array integers2;       // 10-element Array by default

   // print integers1 size and contents
   cout << "Size of array integers1 is "
        << integers1.getSize()
        << "\nArray after initialization:\n" << integers1;

   // print integers2 size and contents
   cout << "\nSize of array integers2 is "
        << integers2.getSize()
        << "\nArray after initialization:\n" << integers2;

```

29

Outline

fig08_06.cpp
(1 of 3)



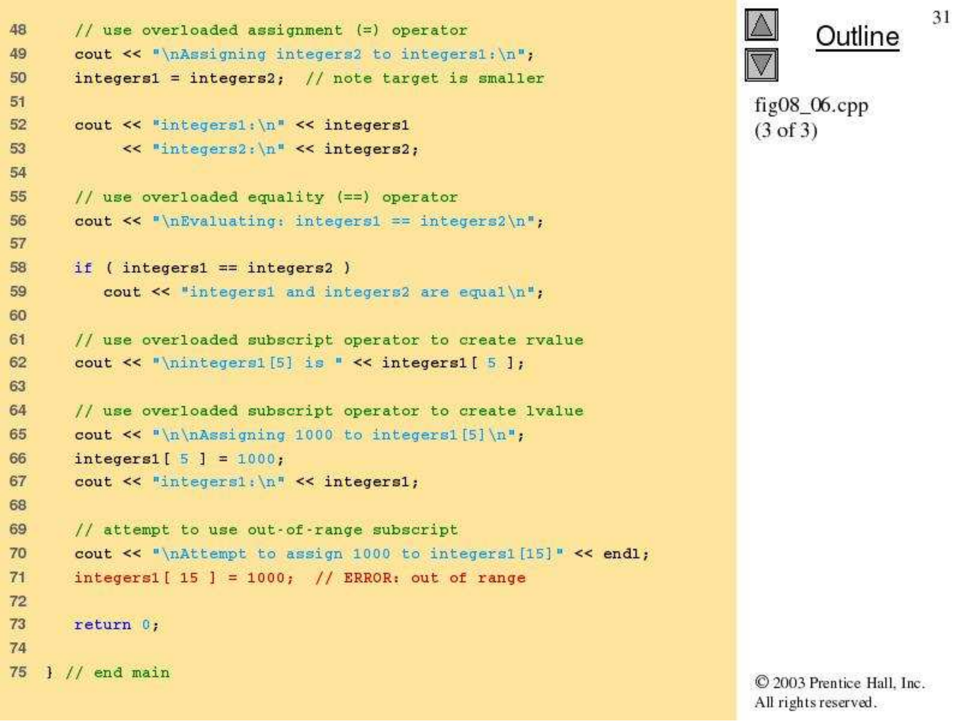
```
   // input and print integers1 and integers2
   cout << "\nInput 17 integers:\n";
   cin >> integers1 >> integers2;

   cout << "\nAfter input, the arrays contain:\n"
        << "integers1:\n" << integers1
        << "integers2:\n" << integers2;

   // use overloaded inequality (!=) operator
   cout << "\nEvaluating: integers1 != integers2\n";

   if ( integers1 != integers2 )
      cout << "integers1 and integers2 are not equal\n";

   // create array integers3 using integers1 as an
   // initializer; print size and contents
   Array integers3( integers1 );  // calls copy constructor

   cout << "\nSize of array integers3 is "
        << integers3.getSize()
        << "\nArray after initialization:\n" << integers3;

```

30

Outline

fig08_06.cpp
(2 of 3)

Figure 3.8: **Array** class test program. (part 1 of 2)



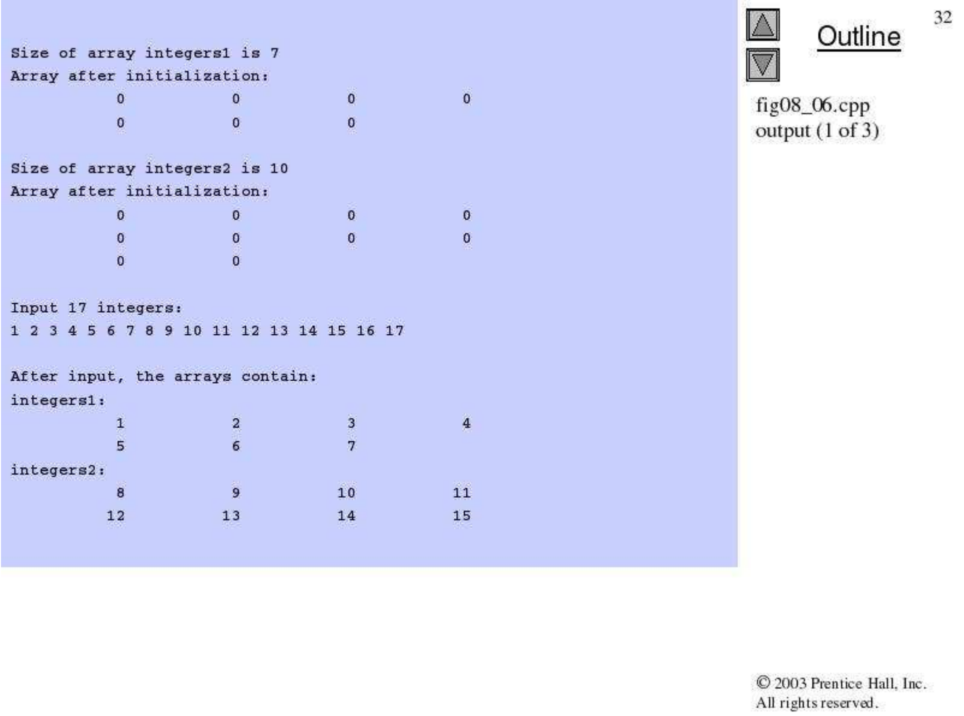
```
   // use overloaded assignment (=) operator
   cout << "\nAssigning integers2 to integers1:\n";
   integers1 = integers2;  // note target is smaller

   cout << "integers1:\n" << integers1
        << "integers2:\n" << integers2;

   // use overloaded equality (==) operator
   cout << "\nEvaluating: integers1 == integers2\n";

   if ( integers1 == integers2 )
      cout << "integers1 and integers2 are equal\n";

   // use overloaded subscript operator to create rvalue
   cout << "\nintegers1[5] is " << integers1[ 5 ];

   // use overloaded subscript operator to create lvalue
   cout << "\n\nAssigning 1000 to integers1[5]\n";
   integers1[ 5 ] = 1000;
   cout << "integers1:\n" << integers1;

   // attempt to use out-of-range subscript
   cout << "\nAttempt to assign 1000 to integers1[15]" << endl;
   integers1[ 15 ] = 1000;  // ERROR: out of range

   return 0;

} // end main
```

31

Outline

fig08_06.cpp
(3 of 3)

Figure 3.9: **Array** class test program. (part 2 of 2)



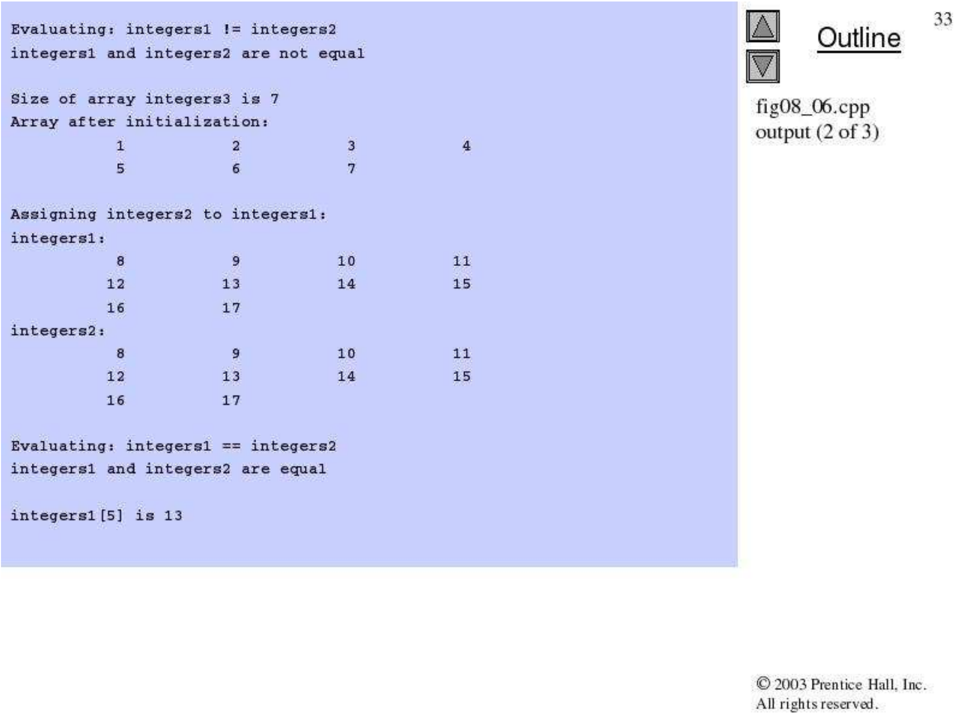
32

Outline

fig08_06.cpp
output (1 of 3)

```
Size of array integers1 is 7
Array after initialization:
           0           0           0           0
           0           0           0

Size of array integers2 is 10
Array after initialization:
           0           0           0           0
           0           0           0           0
           0           0

Input 17 integers:
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17

After input, the arrays contain:
integers1:
           1           2           3           4
           5           6           7
integers2:
           8           9          10          11
          12          13          14          15
```

© 2003 Prentice Hall, Inc.
All rights reserved.



33

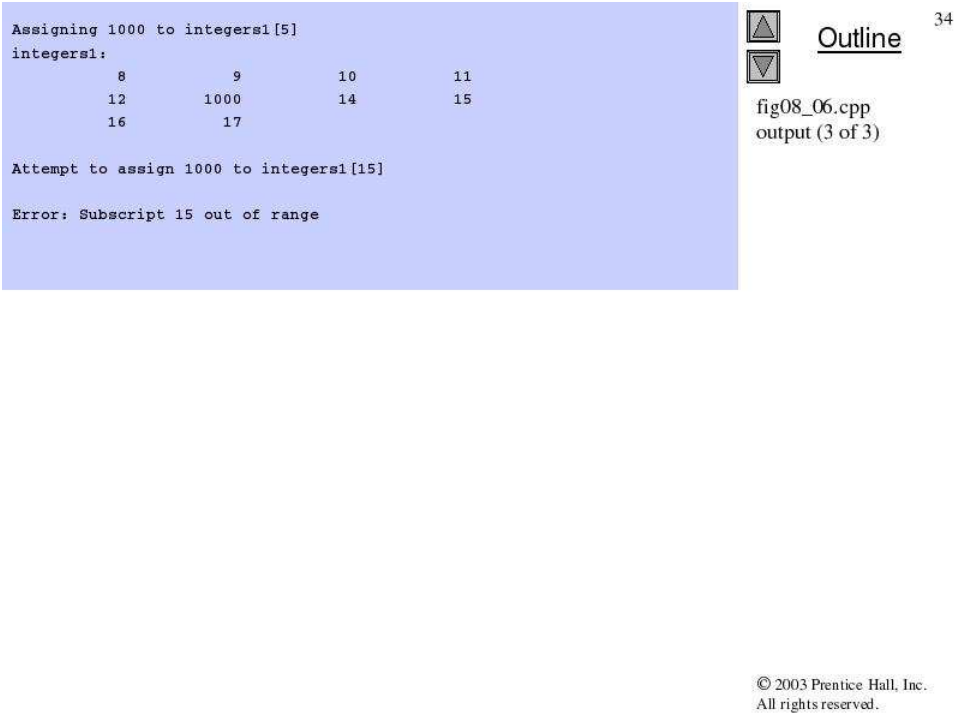
Outline

fig08_06.cpp
output (2 of 3)

```
Evaluating: integers1 != integers2
integers1 and integers2 are not equal

Size of array integers3 is 7
Array after initialization:
           1           2           3           4
           5           6           7

Assigning integers2 to integers1:
integers1:
           8           9          10          11
          12          13          14          15
          16          17
integers2:
           8           9          10          11
          12          13          14          15
          16          17

Evaluating: integers1 == integers2
integers1 and integers2 are equal

integers1[5] is 13
```

© 2003 Prentice Hall, Inc.
All rights reserved.

Figure 3.10: **Array** class test program, output. (part 1 of 2)



```
Assigning 1000 to integers1[5]
integers1:
           8           9          10          11
          12        1000          14          15
          16          17

Attempt to assign 1000 to integers1[15]

Error: Subscript 15 out of range
```

Figure 3.11: **Array** class test program, output. (part 2 of 2)

3.9 Converting between Types

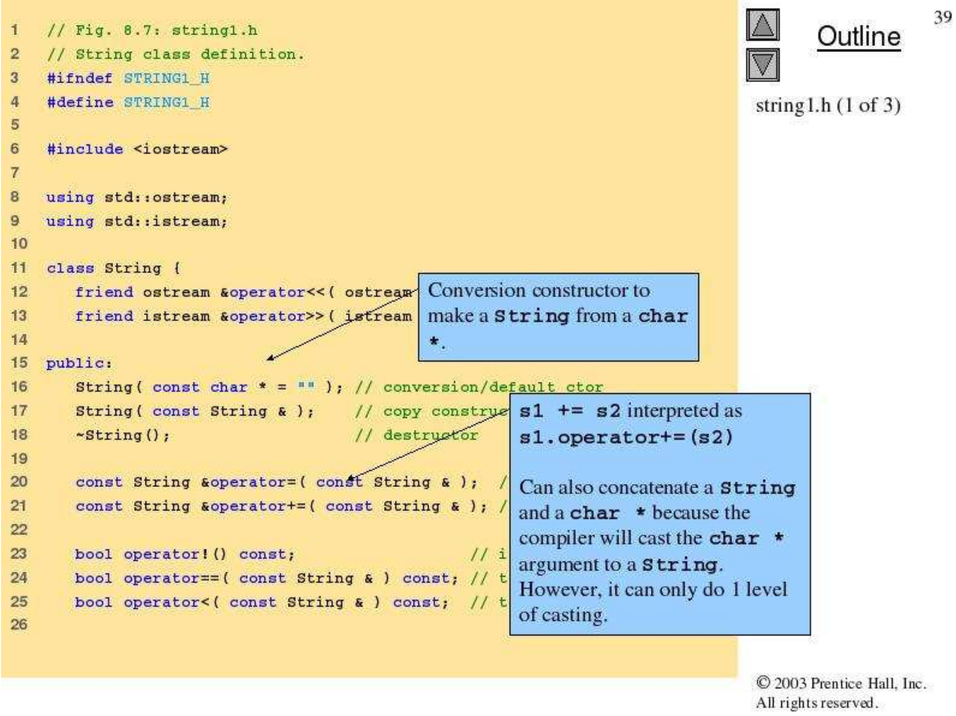
Sometimes all the operations ”stay within a type”.

- Casting
  - Traditionally, cast integers to floats, etc.
  - May need to convert between user-defined types
- Cast operator (conversion operator)
  - Convert from
    - One class to another
    - Class to built-in type (**int**, **char**, etc.)
  - Must be non-**static** member function; Cannot be **friend**
  - Do not specify return type; implicitly returns type to which you are converting
- Example
  - Prototype
    - **A::operator char *() const;**
    - Casts class **A** to a temporary **char ***
    - **(char *)s** calls **s.operator char*()**
  - Also, overloaded cast-operator functions can be defined for converting objects of user-defined types into built-in types or into objects of other user-defined types.
    - **A::operator int() const;**
    - **A::operator OtherClass() const;**
- Casting can prevent need for overloading
  - Suppose class **String** can be cast to **char ***
  - **cout << s; // s is a String**
    - Compiler implicitly converts **s** to **char ***
    - Do not have to overload <<

3.10 Case Study: A String Class

- Build class String
  - To handle String creation, manipulation
  - Class **string** in standard library (more Chapter 15)
- Conversion constructor
  - Single-argument constructor
  - Turns objects of other types into class objects
    - **String s1(”hi”);**
    - Creates a **String** from a **char ***
  - Any single-argument constructor is a conversion constructor

The programs of Figs. 3.12-3.21 demonstrates the building of our own **String** class to handle the creation and manipulation of strings.



39

Outline

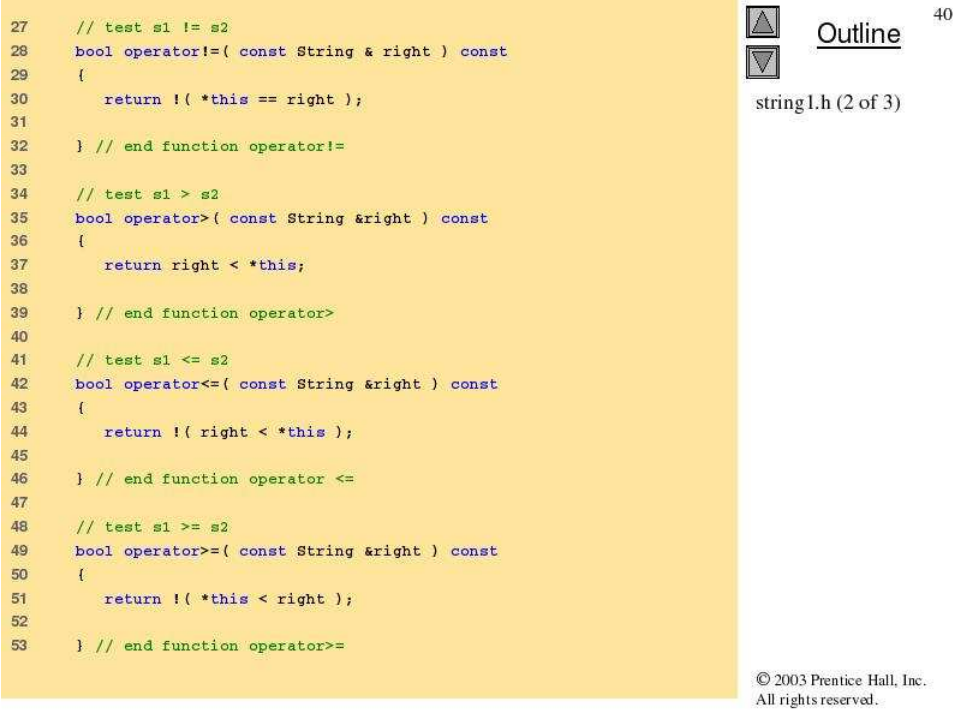
string1.h (1 of 3)

```
// Fig. 8.7: string1.h
// String class definition.
#ifndef STRING1_H
#define STRING1_H

#include <iostream>

using std::ostream;
using std::istream;

class String {
   friend ostream &operator<<( ostream
   friend istream &operator>>( istream

public:
   String( const char * = "" ); // conversion/default ctor
   String( const String & );    // copy construc
   ~String();                   // destructor

   const String &operator=( const String & );  /
   const String &operator+=( const String & ); /

   bool operator!() const;                   // i
   bool operator==( const String & ) const; // t
   bool operator<( const String & ) const;  // t

```

Conversion constructor to make a **String** from a **char ***.

s1 += s2 interpreted as **s1.operator+=(s2)**

Can also concatenate a **String** and a **char *** because the compiler will cast the **char *** argument to a **String**. However, it can only do 1 level of casting.



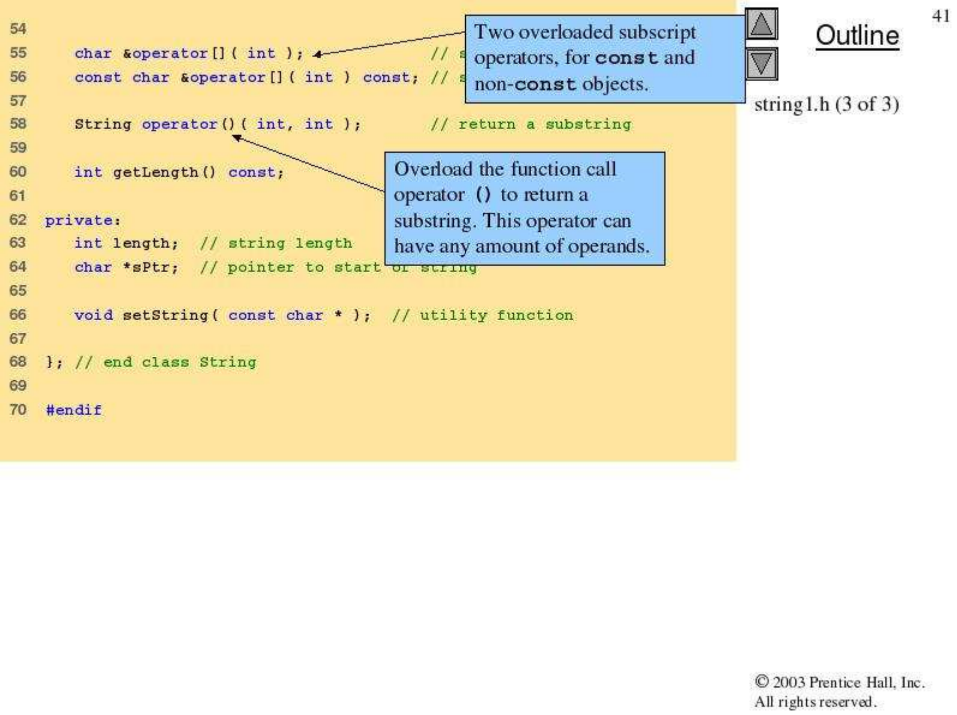
40

Outline

string1.h (2 of 3)

```
   // test s1 != s2
   bool operator!=( const String & right ) const
   {
      return !( *this == right );

   } // end function operator!=

   // test s1 > s2
   bool operator>( const String &right ) const
   {
      return right < *this;

   } // end function operator>

   // test s1 <= s2
   bool operator<=( const String &right ) const
   {
      return !( right < *this );

   } // end function operator <=

   // test s1 >= s2
   bool operator>=( const String &right ) const
   {
      return !( *this < right );

   } // end function operator>=
```

Figure 3.12: **String** class definition with operator overloading. (part 1 of 2)



41

Outline

string1.h (3 of 3)

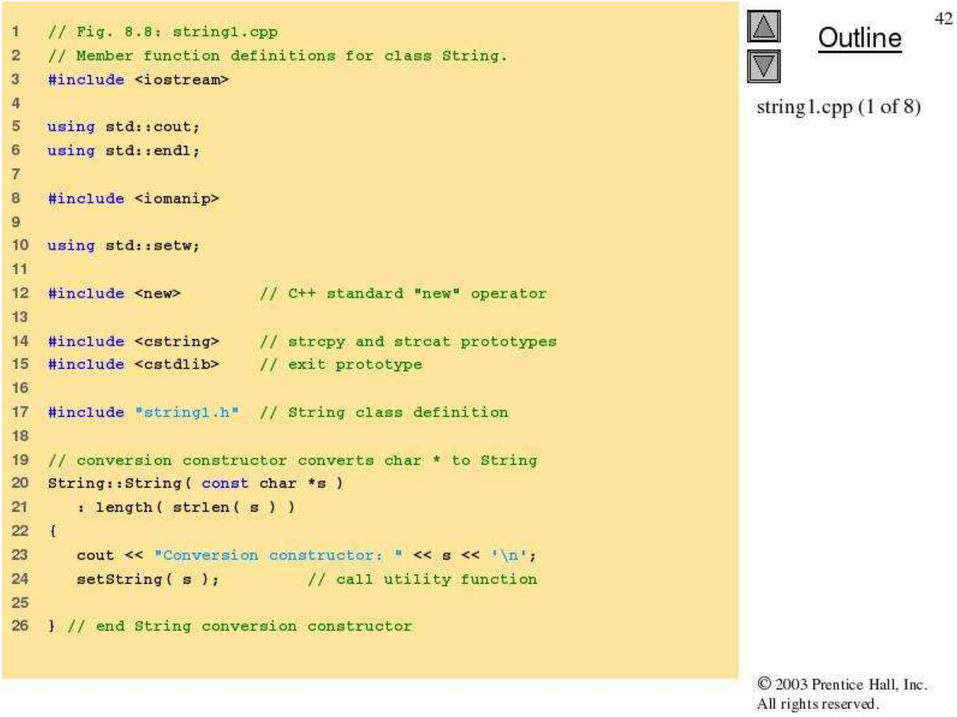
```

   char &operator[]( int );                // s
   const char &operator[]( int ) const; // s

   String operator()( int, int );        // return a substring

   int getLength() const;

private:
   int length;  // string length
   char *sPtr;  // pointer to start of string

   void setString( const char * );  // utility function

}; // end class String

#endif
```

Two overloaded subscript operators, for **const** and non-**const** objects.

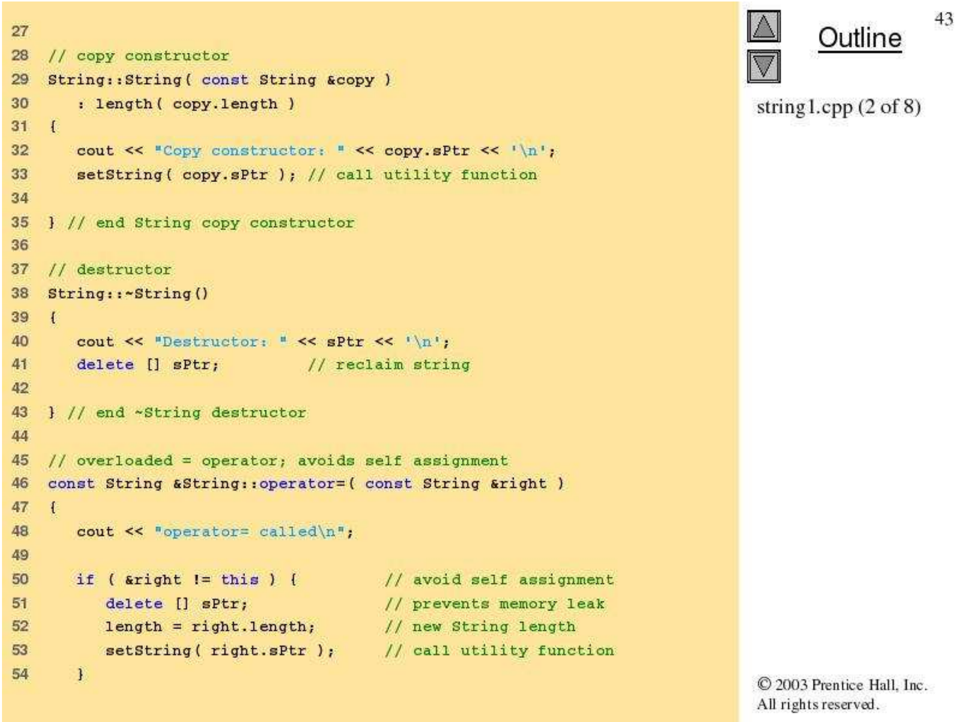
Overload the function call operator **()** to return a substring. This operator can have any amount of operands.

© 2003 Prentice Hall, Inc.
All rights reserved.

Figure 3.13: **String** class definition with operator overloading. (part 2 of 2)

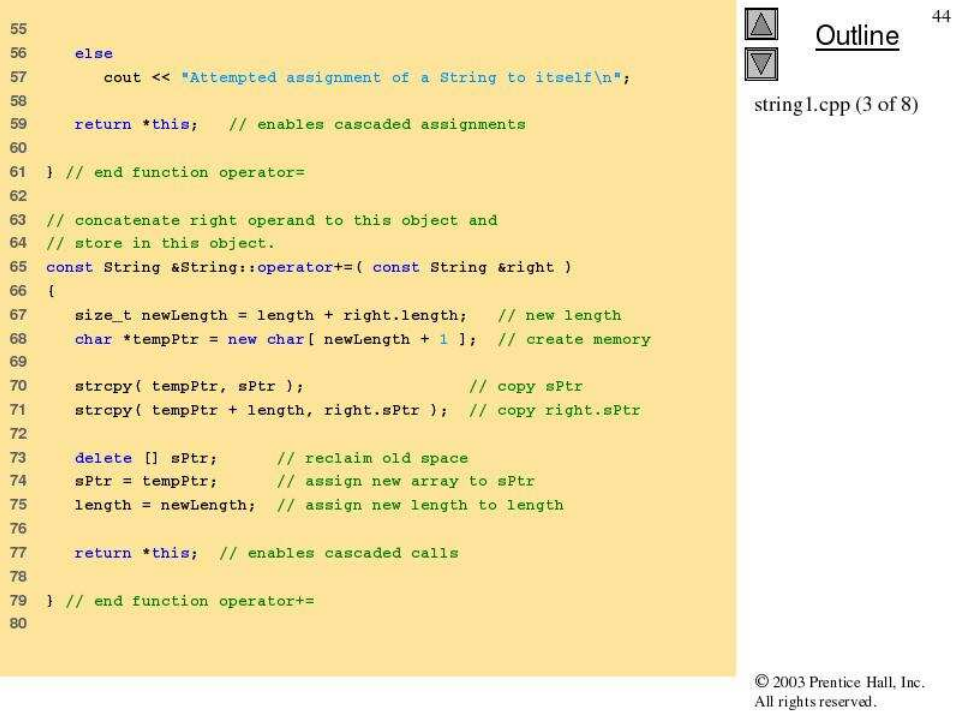


```
// Fig. 8.8: string1.cpp
// Member function definitions for class String.
#include <iostream>

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

#include <iomanip>

using std::setw;

#include <new>        // C++ standard "new" operator

#include <cstring>    // strcpy and strcat prototypes
#include <cstdlib>    // exit prototype

#include "string1.h"  // String class definition

// conversion constructor converts char * to String
String::String( const char *s )
   : length( strlen( s ) )
{
   cout << "Conversion constructor: " << s << '\n';
   setString( s );          // call utility function

} // end String conversion constructor
```



```

// copy constructor
String::String( const String &copy )
   : length( copy.length )
{
   cout << "Copy constructor: " << copy.sPtr << '\n';
   setString( copy.sPtr ); // call utility function

} // end String copy constructor

// destructor
String::~String()
{
   cout << "Destructor: " << sPtr << '\n';
   delete [] sPtr;          // reclaim string

} // end ~String destructor

// overloaded = operator; avoids self assignment
const String &String::operator=( const String &right )
{
   cout << "operator= called\n";

   if ( &right != this ) {         // avoid self assignment
      delete [] sPtr;              // prevents memory leak
      length = right.length;       // new String length
      setString( right.sPtr );     // call utility function
   }
```

Figure 3.14: **String** class member-function and **friend**-function definition. (part 1 of 4)

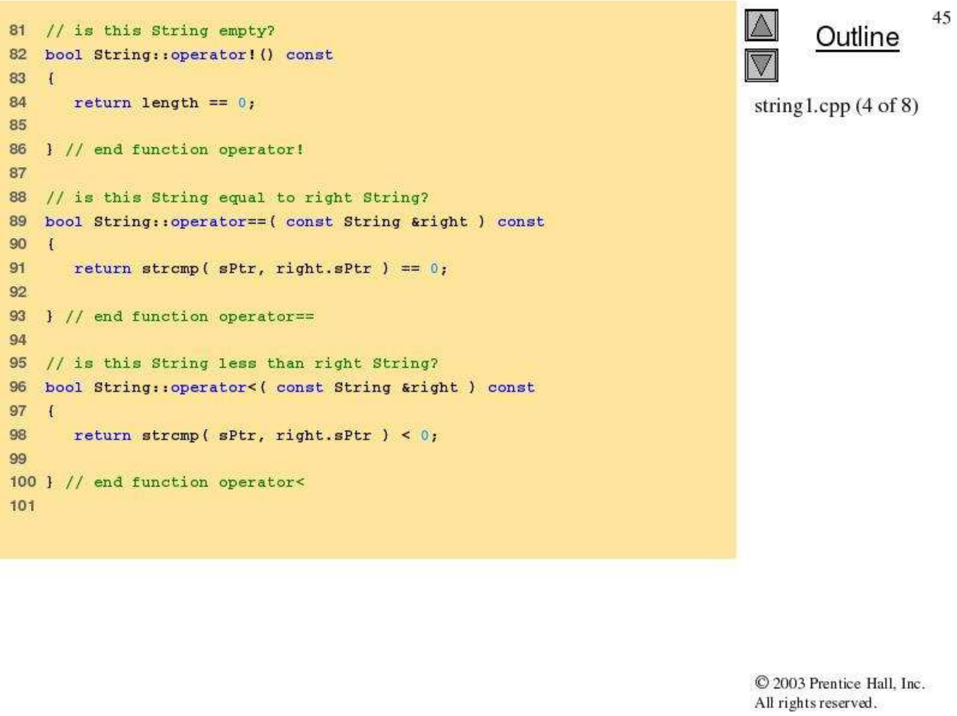


44

Outline

string1.cpp (3 of 8)

```

   else
      cout << "Attempted assignment of a String to itself\n";

   return *this;   // enables cascaded assignments

} // end function operator=

// concatenate right operand to this object and
// store in this object.
const String &String::operator+=( const String &right )
{
   size_t newLength = length + right.length;   // new length
   char *tempPtr = new char[ newLength + 1 ];  // create memory

   strcpy( tempPtr, sPtr );                  // copy sPtr
   strcpy( tempPtr + length, right.sPtr );   // copy right.sPtr

   delete [] sPtr;      // reclaim old space
   sPtr = tempPtr;      // assign new array to sPtr
   length = newLength;  // assign new length to length

   return *this;  // enables cascaded calls

} // end function operator+=

```



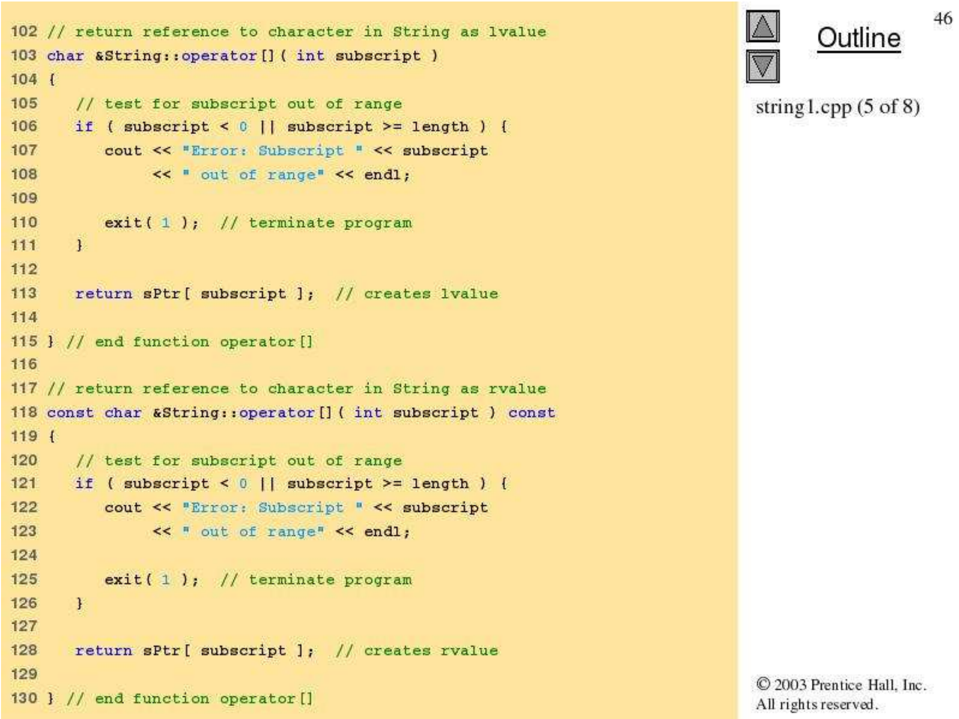
45

Outline

string1.cpp (4 of 8)

```
// is this String empty?
bool String::operator!() const
{
   return length == 0;

} // end function operator!

// is this String equal to right String?
bool String::operator==( const String &right ) const
{
   return strcmp( sPtr, right.sPtr ) == 0;

} // end function operator==

// is this String less than right String?
bool String::operator<( const String &right ) const
{
   return strcmp( sPtr, right.sPtr ) < 0;

} // end function operator<

```

Figure 3.15: **String** class member-function and **friend**-function definition. (part 2 of 4)

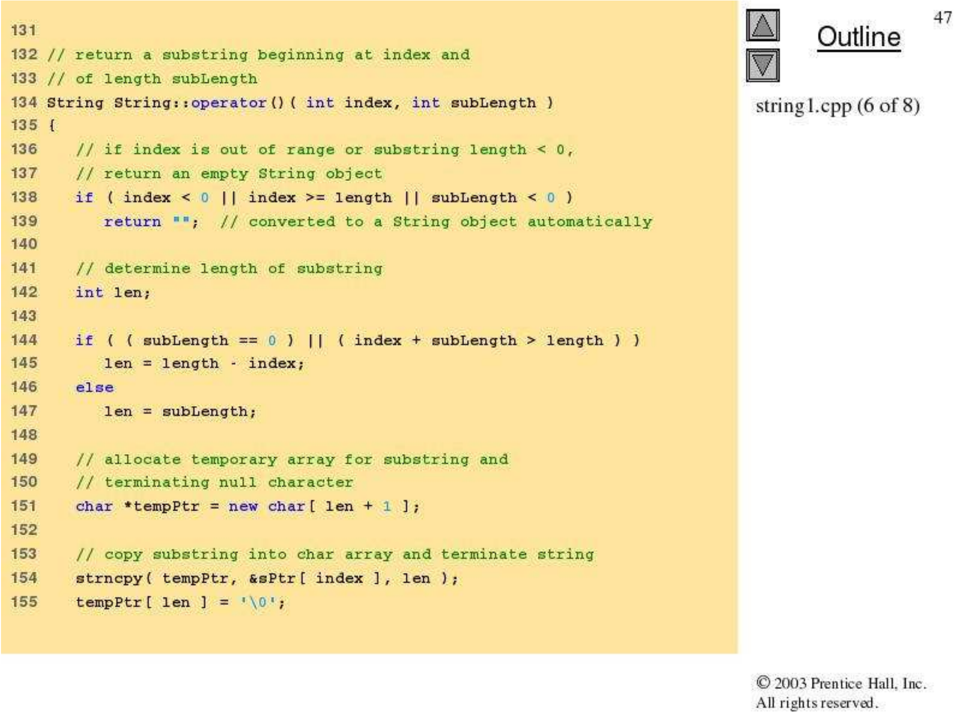


```cpp
// return reference to character in String as lvalue
char &String::operator[]( int subscript )
{
   // test for subscript out of range
   if ( subscript < 0 || subscript >= length ) {
      cout << "Error: Subscript " << subscript
           << " out of range" << endl;

      exit( 1 );  // terminate program
   }

   return sPtr[ subscript ];  // creates lvalue

} // end function operator[]

// return reference to character in String as rvalue
const char &String::operator[]( int subscript ) const
{
   // test for subscript out of range
   if ( subscript < 0 || subscript >= length ) {
      cout << "Error: Subscript " << subscript
           << " out of range" << endl;

      exit( 1 );  // terminate program
   }

   return sPtr[ subscript ];  // creates rvalue

} // end function operator[]
```

46

Outline

string1.cpp (5 of 8)



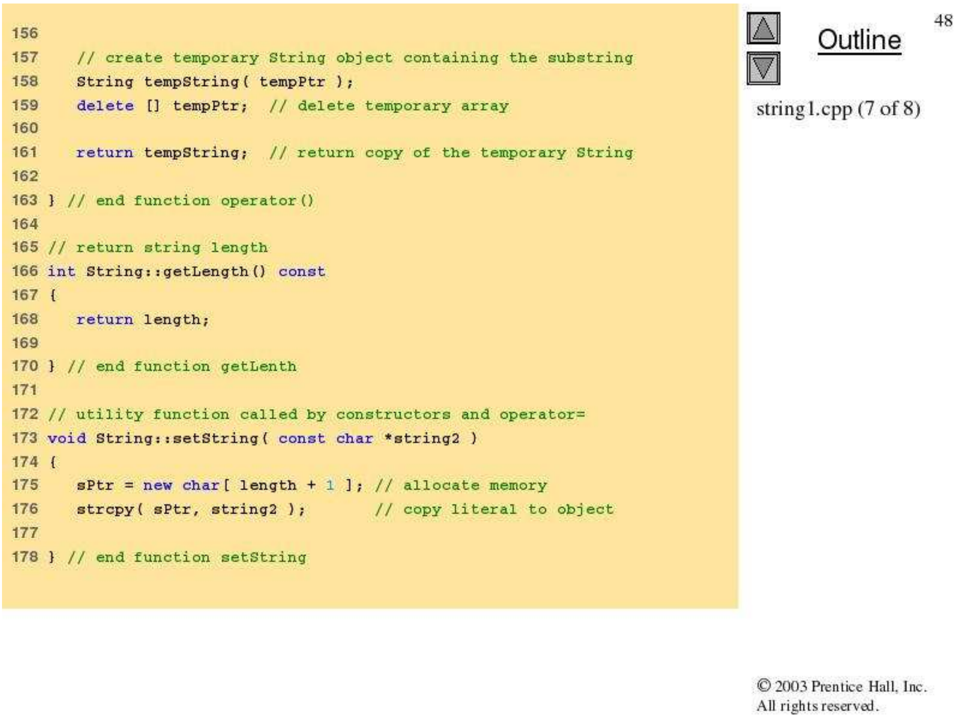
```cpp

// return a substring beginning at index and
// of length subLength
String String::operator()( int index, int subLength )
{
   // if index is out of range or substring length < 0,
   // return an empty String object
   if ( index < 0 || index >= length || subLength < 0 )
      return "";  // converted to a String object automatically

   // determine length of substring
   int len;

   if ( ( subLength == 0 ) || ( index + subLength > length ) )
      len = length - index;
   else
      len = subLength;

   // allocate temporary array for substring and
   // terminating null character
   char *tempPtr = new char[ len + 1 ];

   // copy substring into char array and terminate string
   strncpy( tempPtr, &sPtr[ index ], len );
   tempPtr[ len ] = '\0';
```

47

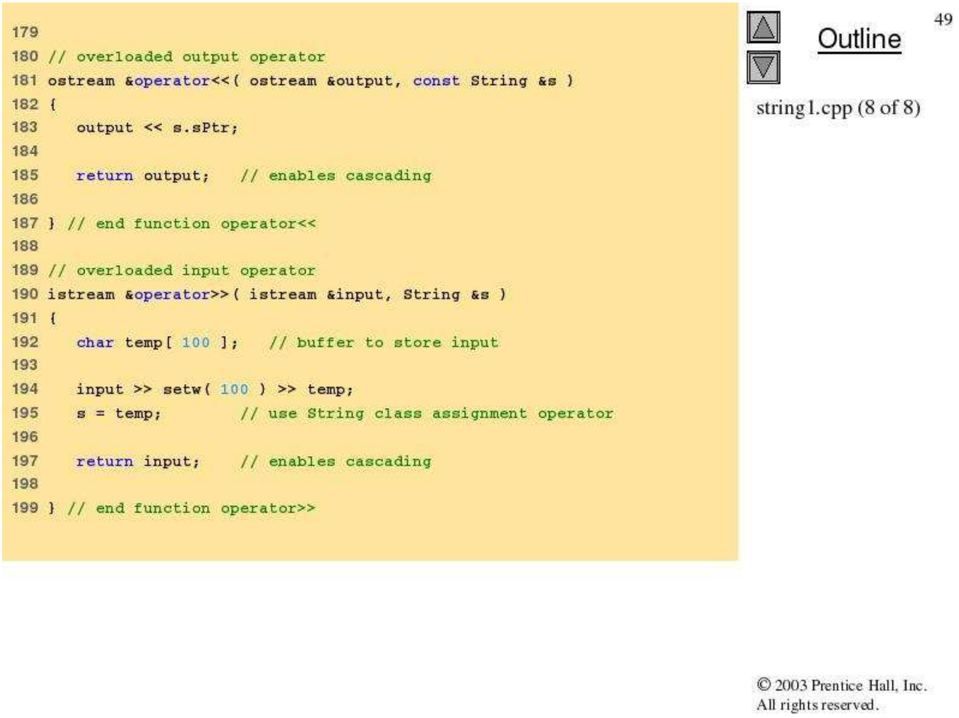
Outline

string1.cpp (6 of 8)

Figure 3.16: **String** class member-function and **friend**-function definition. (part 3 of 4)

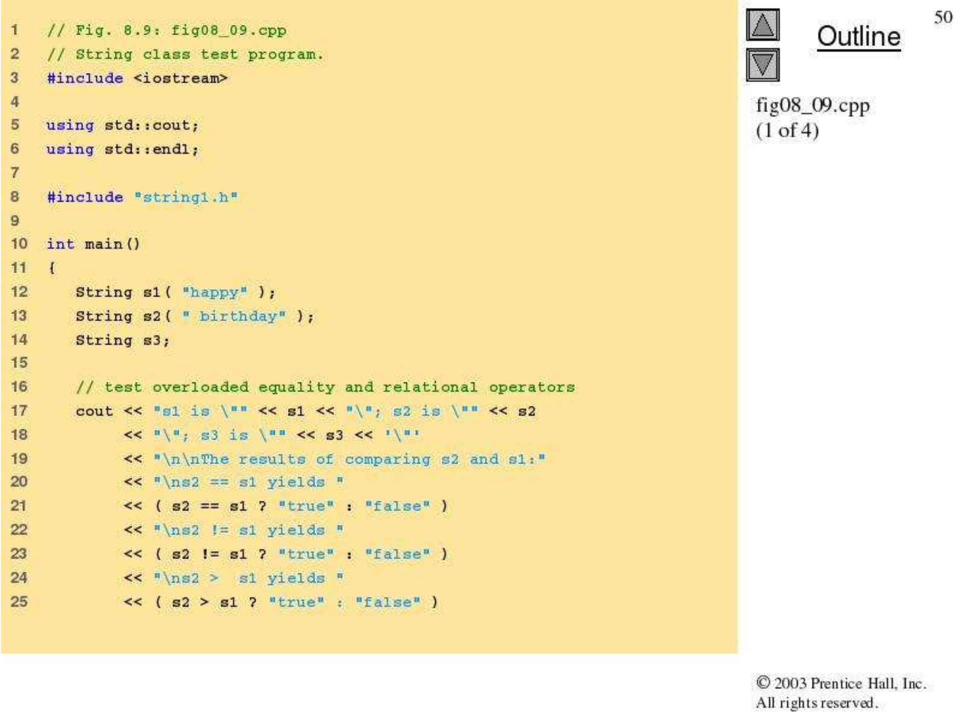


```

   // create temporary String object containing the substring
   String tempString( tempPtr );
   delete [] tempPtr;  // delete temporary array

   return tempString;  // return copy of the temporary String

} // end function operator()

// return string length
int String::getLength() const
{
   return length;

} // end function getLenth

// utility function called by constructors and operator=
void String::setString( const char *string2 )
{
   sPtr = new char[ length + 1 ]; // allocate memory
   strcpy( sPtr, string2 );       // copy literal to object

} // end function setString
```



```

// overloaded output operator
ostream &operator<<( ostream &output, const String &s )
{
   output << s.sPtr;

   return output;   // enables cascading

} // end function operator<<

// overloaded input operator
istream &operator>>( istream &input, String &s )
{
   char temp[ 100 ];   // buffer to store input

   input >> setw( 100 ) >> temp;
   s = temp;         // use String class assignment operator

   return input;    // enables cascading

} // end function operator>>
```

Figure 3.17: **String** class member-function and **friend**-function definition. (part 4 of 4)

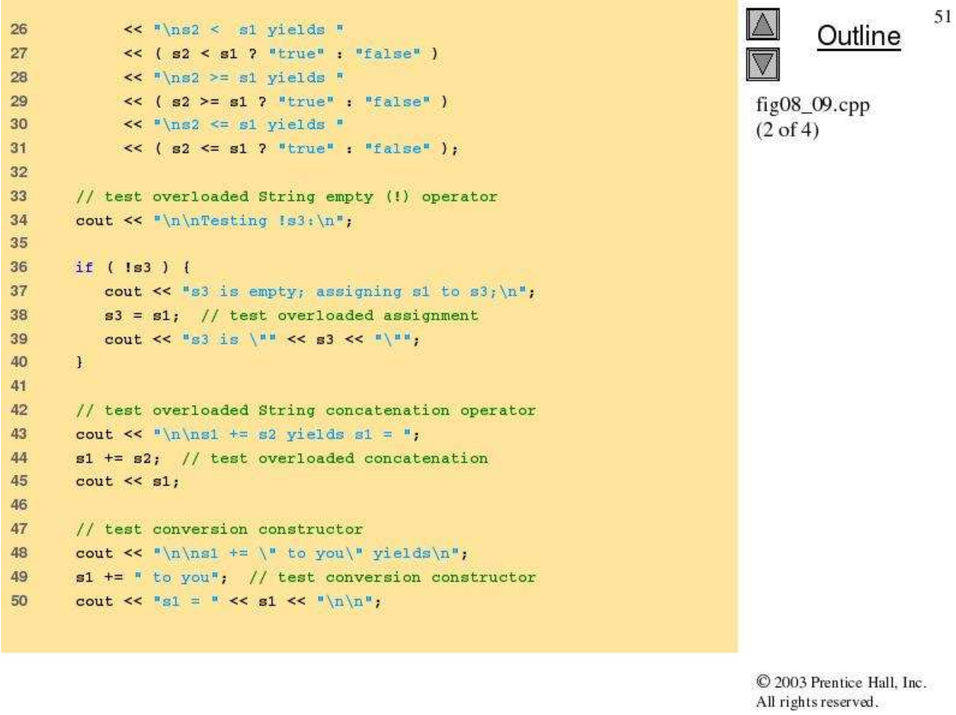


```
// Fig. 8.9: fig08_09.cpp
// String class test program.
#include <iostream>

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

#include "string1.h"

int main()
{
   String s1( "happy" );
   String s2( " birthday" );
   String s3;

   // test overloaded equality and relational operators
   cout << "s1 is \"" << s1 << "\"; s2 is \"" << s2
        << "\"; s3 is \"" << s3 << '\"'
        << "\n\nThe results of comparing s2 and s1:"
        << "\ns2 == s1 yields "
        << ( s2 == s1 ? "true" : "false" )
        << "\ns2 != s1 yields "
        << ( s2 != s1 ? "true" : "false" )
        << "\ns2 >  s1 yields "
        << ( s2 > s1 ? "true" : "false" )
```

50

Outline

fig08_09.cpp
(1 of 4)



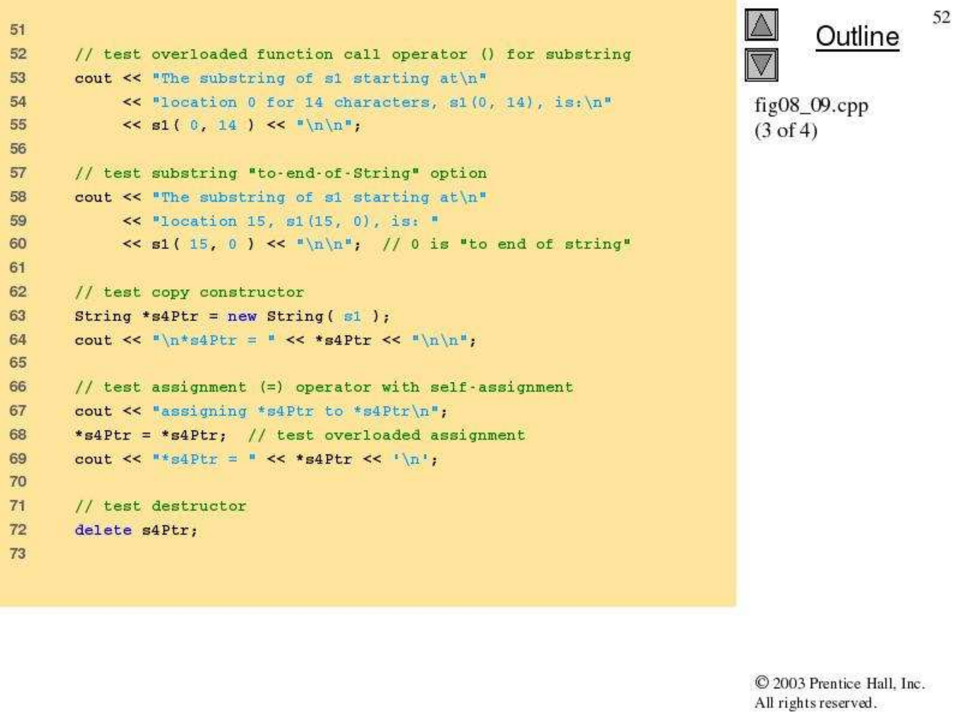
```
        << "\ns2 <  s1 yields "
        << ( s2 < s1 ? "true" : "false" )
        << "\ns2 >= s1 yields "
        << ( s2 >= s1 ? "true" : "false" )
        << "\ns2 <= s1 yields "
        << ( s2 <= s1 ? "true" : "false" );

   // test overloaded String empty (!) operator
   cout << "\n\nTesting !s3:\n";

   if ( !s3 ) {
      cout << "s3 is empty; assigning s1 to s3;\n";
      s3 = s1;  // test overloaded assignment
      cout << "s3 is \"" << s3 << "\"";
   }

   // test overloaded String concatenation operator
   cout << "\n\ns1 += s2 yields s1 = ";
   s1 += s2;  // test overloaded concatenation
   cout << s1;

   // test conversion constructor
   cout << "\n\ns1 += \" to you\" yields\n";
   s1 += " to you";  // test conversion constructor
   cout << "s1 = " << s1 << "\n\n";
```

51

Outline

fig08_09.cpp
(2 of 4)

Figure 3.18: **String** class test program. (part 1 of 2)



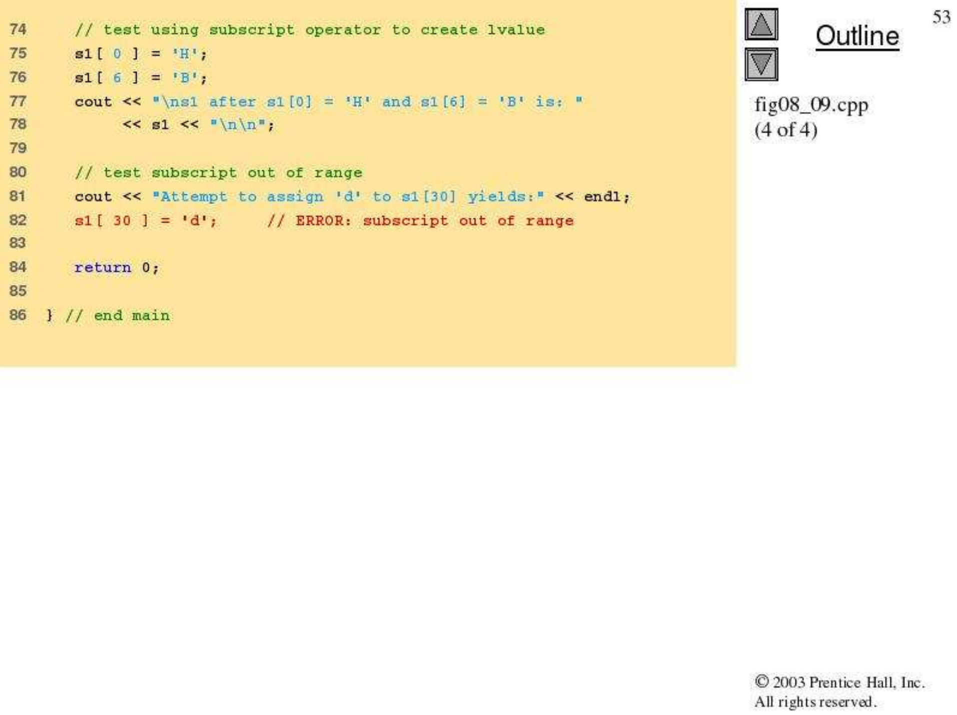
```

   // test overloaded function call operator () for substring
   cout << "The substring of s1 starting at\n"
        << "location 0 for 14 characters, s1(0, 14), is:\n"
        << s1( 0, 14 ) << "\n\n";

   // test substring "to-end-of-String" option
   cout << "The substring of s1 starting at\n"
        << "location 15, s1(15, 0), is: "
        << s1( 15, 0 ) << "\n\n";  // 0 is "to end of string"

   // test copy constructor
   String *s4Ptr = new String( s1 );
   cout << "\n*s4Ptr = " << *s4Ptr << "\n\n";

   // test assignment (=) operator with self-assignment
   cout << "assigning *s4Ptr to *s4Ptr\n";
   *s4Ptr = *s4Ptr;  // test overloaded assignment
   cout << "*s4Ptr = " << *s4Ptr << '\n';

   // test destructor
   delete s4Ptr;

```

52

Outline

fig08_09.cpp
(3 of 4)

© 2003 Prentice Hall, Inc.
All rights reserved.



```
   // test using subscript operator to create lvalue
   s1[ 0 ] = 'H';
   s1[ 6 ] = 'B';
   cout << "\ns1 after s1[0] = 'H' and s1[6] = 'B' is: "
        << s1 << "\n\n";

   // test subscript out of range
   cout << "Attempt to assign 'd' to s1[30] yields:" << endl;
   s1[ 30 ] = 'd';      // ERROR: subscript out of range

   return 0;

} // end main
```

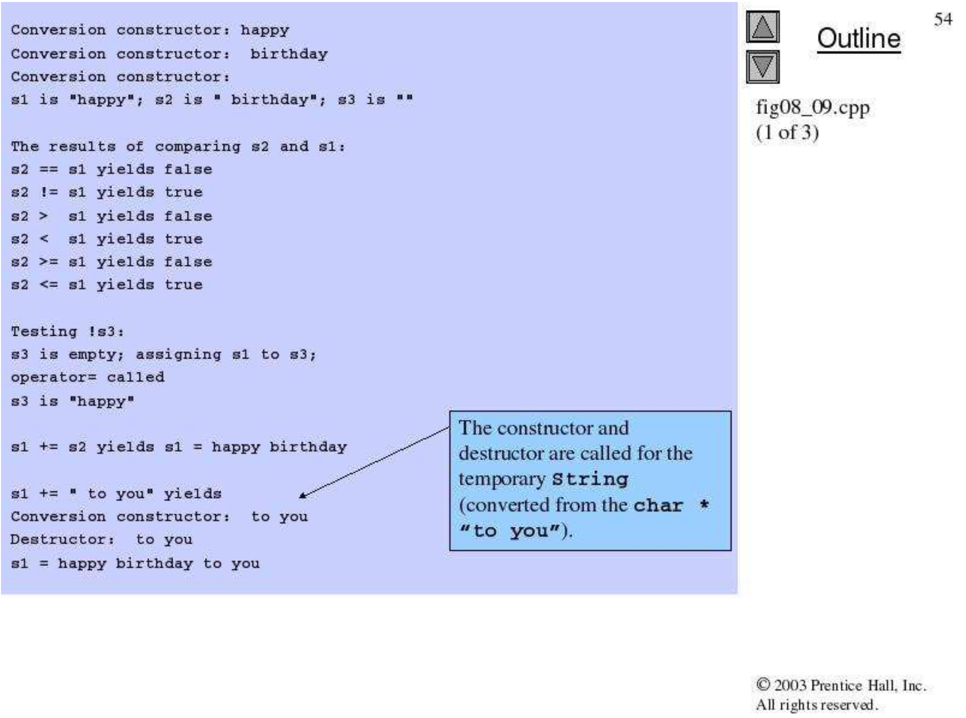
53

Outline

fig08_09.cpp
(4 of 4)

© 2003 Prentice Hall, Inc.
All rights reserved.

Figure 3.19: **String** class test program. (part 2 of 2)



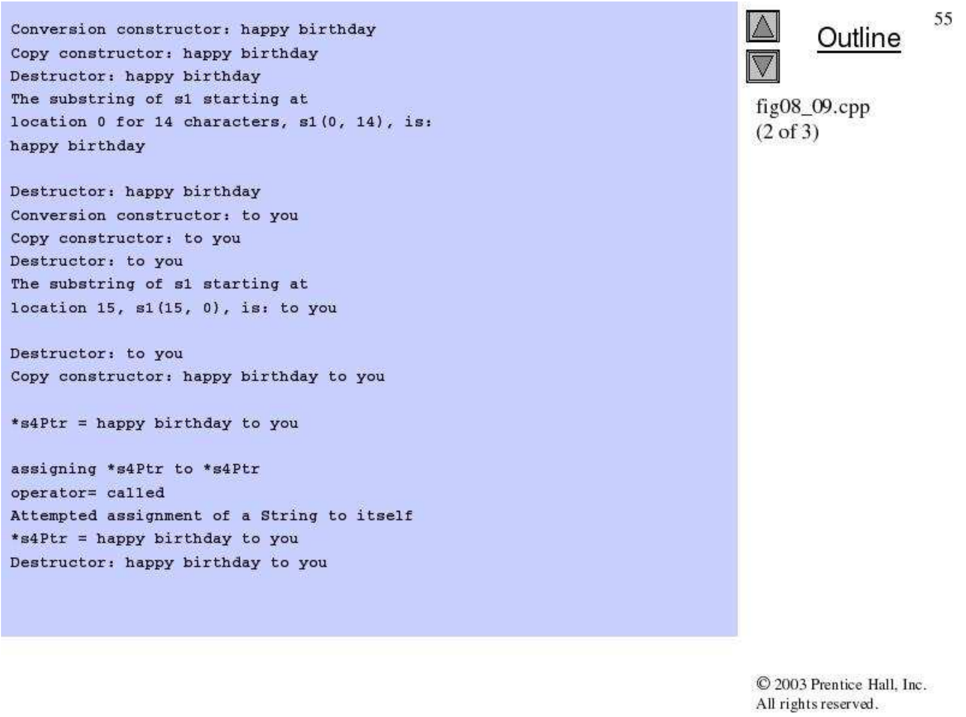
```
Conversion constructor: happy
Conversion constructor:  birthday
Conversion constructor:
s1 is "happy"; s2 is " birthday"; s3 is ""

The results of comparing s2 and s1:
s2 == s1 yields false
s2 != s1 yields true
s2 >  s1 yields false
s2 <  s1 yields true
s2 >= s1 yields false
s2 <= s1 yields true

Testing !s3:
s3 is empty; assigning s1 to s3;
operator= called
s3 is "happy"

s1 += s2 yields s1 = happy birthday

s1 += " to you" yields
Conversion constructor:  to you
Destructor:  to you
s1 = happy birthday to you
```

The constructor and destructor are called for the temporary **String** (converted from the **char *** **"to you"**).

Outline

fig08_09.cpp
(1 of 3)

54



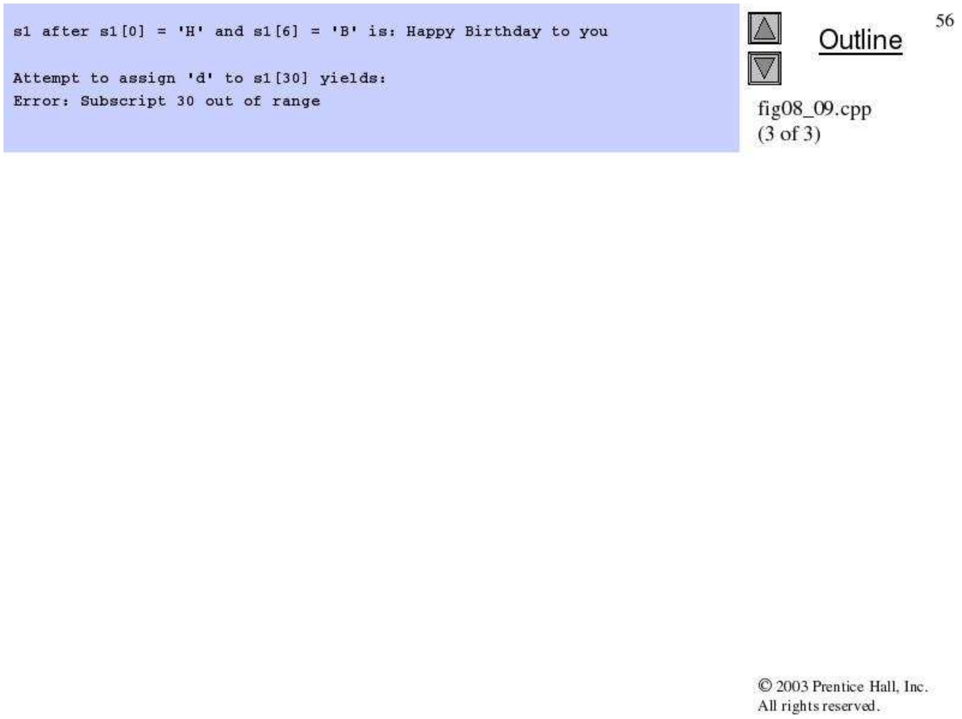
```
Conversion constructor: happy birthday
Copy constructor: happy birthday
Destructor: happy birthday
The substring of s1 starting at
location 0 for 14 characters, s1(0, 14), is:
happy birthday

Destructor: happy birthday
Conversion constructor: to you
Copy constructor: to you
Destructor: to you
The substring of s1 starting at
location 15, s1(15, 0), is: to you

Destructor: to you
Copy constructor: happy birthday to you

*s4Ptr = happy birthday to you

assigning *s4Ptr to *s4Ptr
operator= called
Attempted assignment of a String to itself
*s4Ptr = happy birthday to you
Destructor: happy birthday to you
```

Outline

fig08_09.cpp
(2 of 3)

55

Figure 3.20: **String** class test program, output. (part 1 of 2)



```
s1 after s1[0] = 'H' and s1[6] = 'B' is: Happy Birthday to you

Attempt to assign 'd' to s1[30] yields:
Error: Subscript 30 out of range
```

Figure 3.21: **String** class test program, output. (part 2 of 2)

3.11 Overloading ++ and –

- Increment/decrement operators can be overloaded
  - Add 1 to a **Date** object, **d1**
  - Prototype (member function)
    - **Date &operator++();**
    - **++d1** same as **d1.operator++()**
  - Prototype (non-member)
    - **Friend Date &operator++(Date &);**
    - **++d1** same as **operator++(d1)**
- To distinguish pre/post increment
  - Post increment has a dummy parameter; **int** of **0**
  - Prototype (member function)
    - **Date operator++(int);**

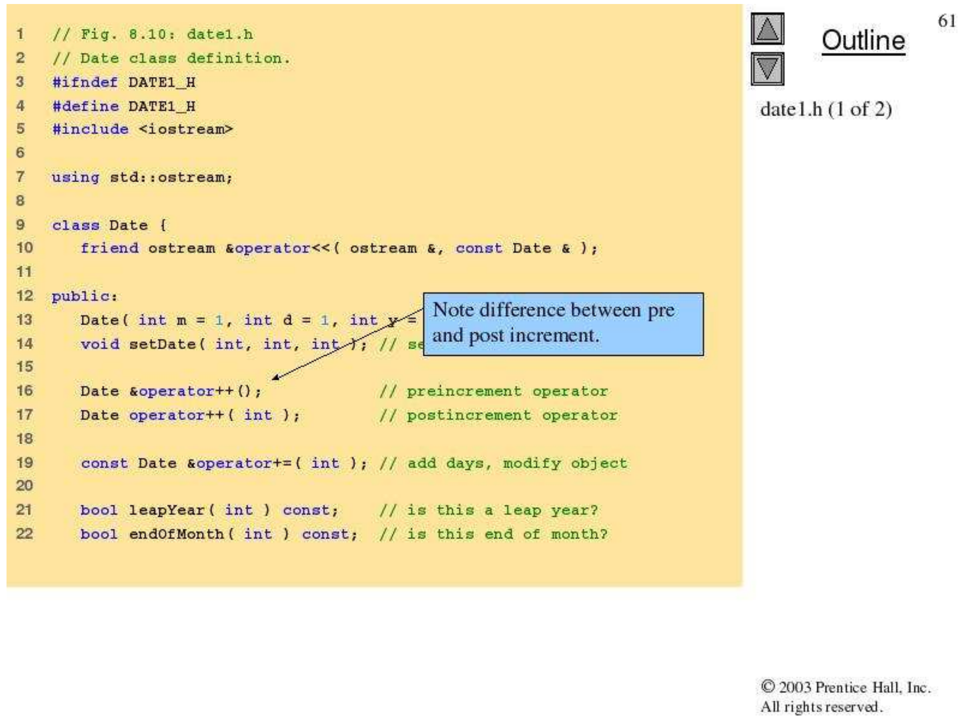
    - **d1++** same as **d1.operator++(0)**
  - Prototype (non-member)
    - **friend Date operator++(Data &, int);**
    - **d1++** same as **operator++(d1, 0)**
  - Integer parameter does not have a name; not even in function definition

- Return values
  - Preincrement
    - Returns by reference (**Date &**)
    - lvalue (can be assigned)
  - Postincrement
    - Returns by value
    - Returns temporary object with old value
    - rvalue (cannot be on left side of assignment)

- Decrement operator analogous

3.12 Case Study: A Date Class

- Example Date class. The class uses overloaded preincrement and postincrement operators to add 1 to the day in a **Date** object, while causing appropriate increments to the month and year if necessary.
  - Overloaded increment operator; Change day, month and year
  - Overloaded += operator
  - Function to test for leap years
  - Function to determine if day is last of month



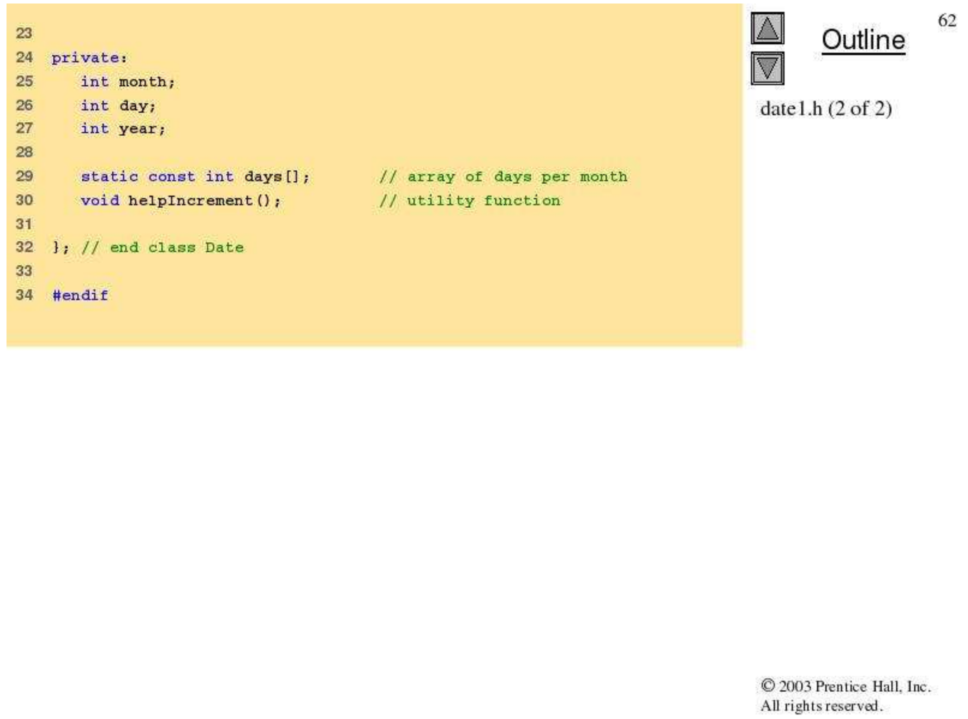
61

Outline

date1.h (1 of 2)

```
// Fig. 8.10: date1.h
// Date class definition.
#ifndef DATE1_H
#define DATE1_H
#include <iostream>

using std::ostream;

class Date {
   friend ostream &operator<<( ostream &, const Date & );

public:
   Date( int m = 1, int d = 1, int y =
   void setDate( int, int, int ); // se

   Date &operator++();              // preincrement operator
   Date operator++( int );          // postincrement operator

   const Date &operator+=( int ); // add days, modify object

   bool leapYear( int ) const;    // is this a leap year?
   bool endOfMonth( int ) const;  // is this end of month?
```

Note difference between pre and post increment.



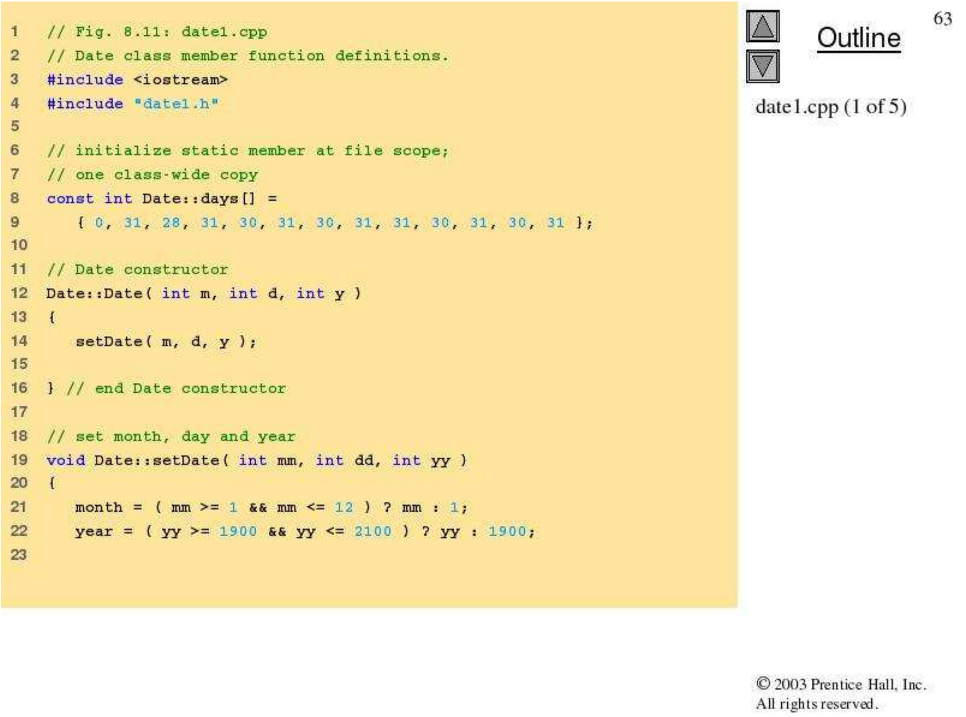
62

Outline

date1.h (2 of 2)

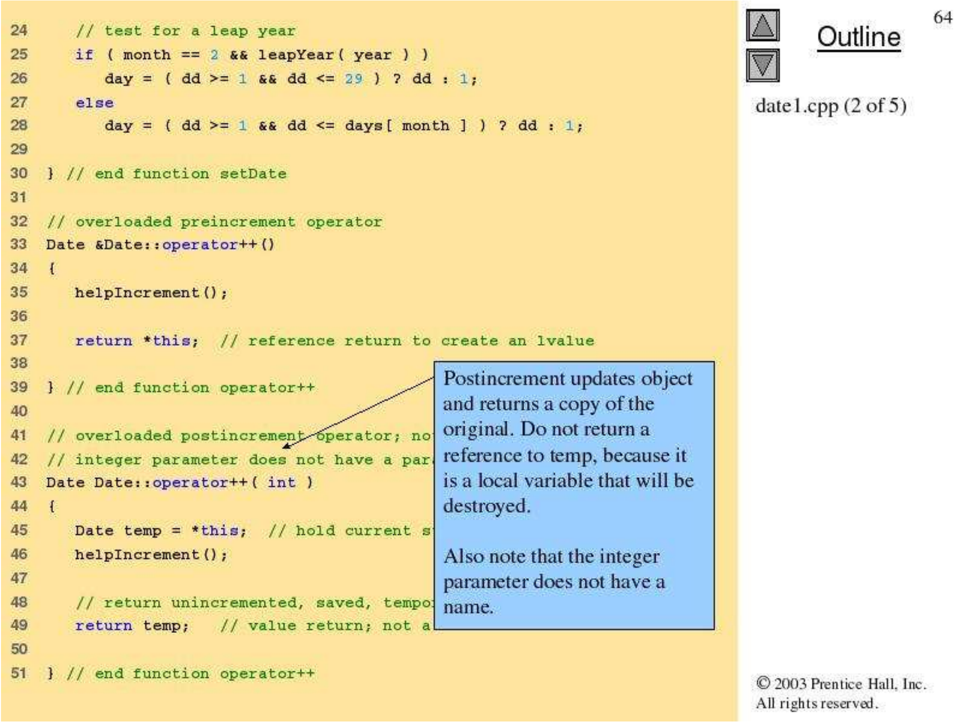
```

private:
   int month;
   int day;
   int year;

   static const int days[];       // array of days per month
   void helpIncrement();          // utility function

}; // end class Date

#endif
```

Figure 3.22: **Date** class definition with overloaded increment operator.



```
// Fig. 8.11: date1.cpp
// Date class member function definitions.
#include <iostream>
#include "date1.h"

// initialize static member at file scope;
// one class-wide copy
const int Date::days[] =
   { 0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 };

// Date constructor
Date::Date( int m, int d, int y )
{
   setDate( m, d, y );

} // end Date constructor

// set month, day and year
void Date::setDate( int mm, int dd, int yy )
{
   month = ( mm >= 1 && mm <= 12 ) ? mm : 1;
   year = ( yy >= 1900 && yy <= 2100 ) ? yy : 1900;

```

© 2003 Prentice Hall, Inc.
All rights reserved.



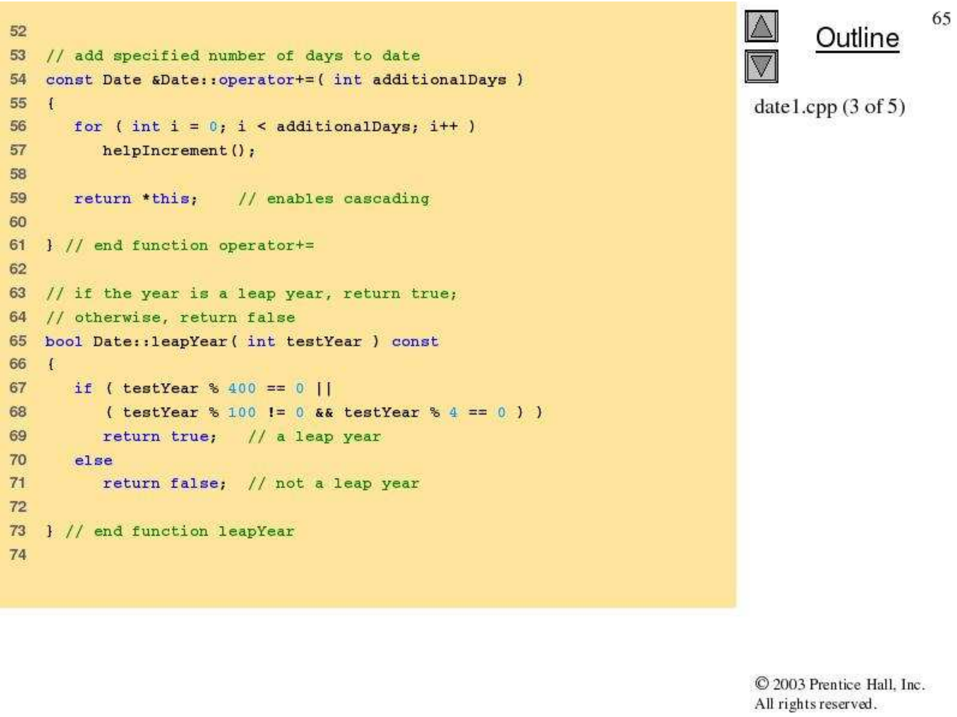
```
   // test for a leap year
   if ( month == 2 && leapYear( year ) )
      day = ( dd >= 1 && dd <= 29 ) ? dd : 1;
   else
      day = ( dd >= 1 && dd <= days[ month ] ) ? dd : 1;

} // end function setDate

// overloaded preincrement operator
Date &Date::operator++()
{
   helpIncrement();

   return *this;  // reference return to create an lvalue

} // end function operator++

// overloaded postincrement operator; no
// integer parameter does not have a par
Date Date::operator++( int )
{
   Date temp = *this;  // hold current s
   helpIncrement();

   // return unincremented, saved, tempo
   return temp;   // value return; not a

} // end function operator++
```

Postincrement updates object and returns a copy of the original. Do not return a reference to temp, because it is a local variable that will be destroyed.

Also note that the integer parameter does not have a name.

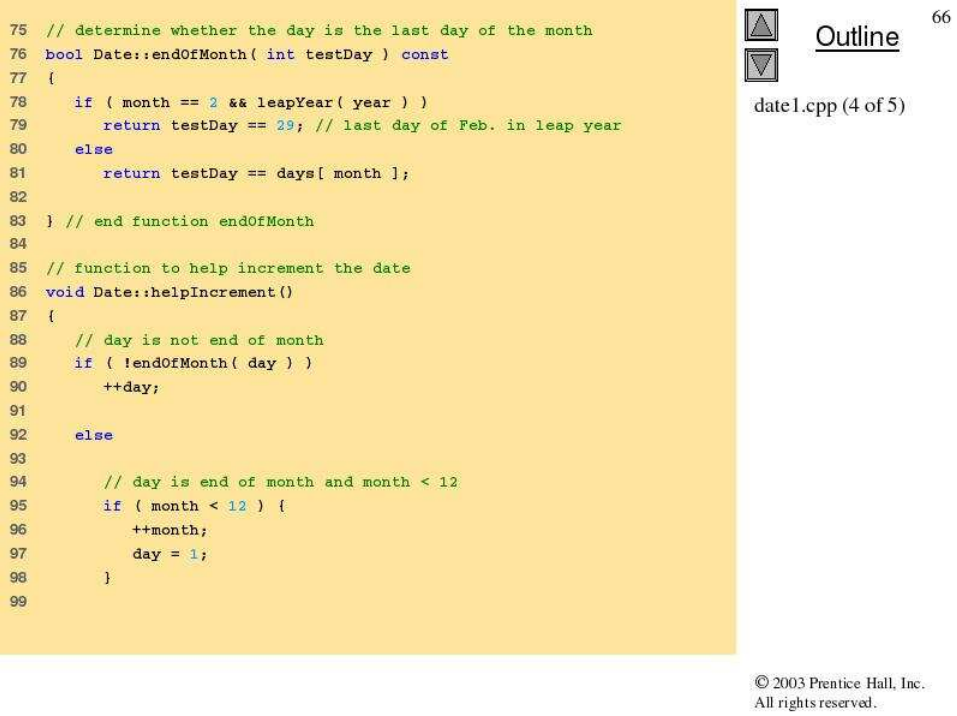
© 2003 Prentice Hall, Inc.
All rights reserved.

Figure 3.23: **Date** class member-and **friend**-function definition. (part 1 of 3)



```cpp

// add specified number of days to date
const Date &Date::operator+=( int additionalDays )
{
   for ( int i = 0; i < additionalDays; i++ )
      helpIncrement();

   return *this;    // enables cascading

} // end function operator+=

// if the year is a leap year, return true;
// otherwise, return false
bool Date::leapYear( int testYear ) const
{
   if ( testYear % 400 == 0 ||
      ( testYear % 100 != 0 && testYear % 4 == 0 ) )
      return true;   // a leap year
   else
      return false;  // not a leap year

} // end function leapYear

```

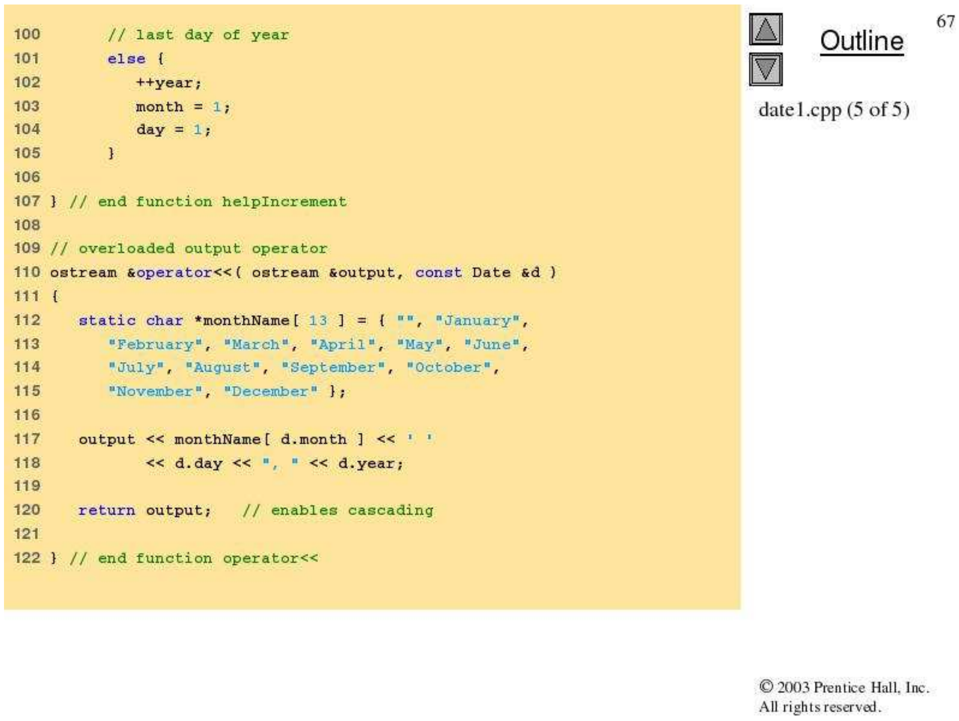
© 2003 Prentice Hall, Inc.
All rights reserved.



```cpp
// determine whether the day is the last day of the month
bool Date::endOfMonth( int testDay ) const
{
   if ( month == 2 && leapYear( year ) )
      return testDay == 29; // last day of Feb. in leap year
   else
      return testDay == days[ month ];

} // end function endOfMonth

// function to help increment the date
void Date::helpIncrement()
{
   // day is not end of month
   if ( !endOfMonth( day ) )
      ++day;

   else

      // day is end of month and month < 12
      if ( month < 12 ) {
         ++month;
         day = 1;
      }

```

© 2003 Prentice Hall, Inc.
All rights reserved.

Figure 3.24: **Date** class member-and **friend**-function definition. (part 2 of 3)



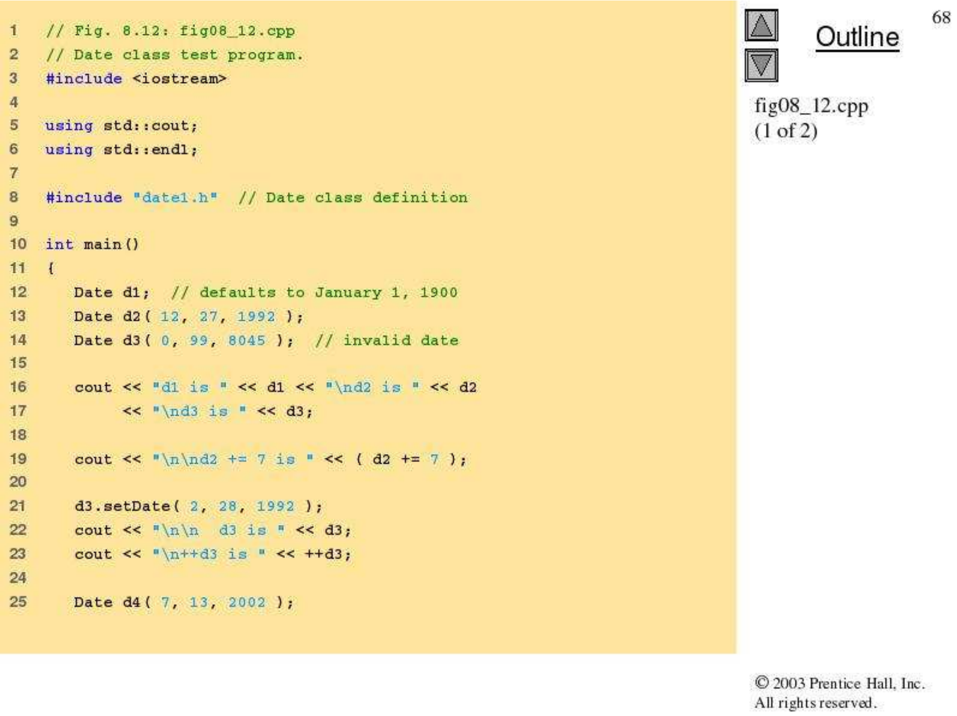
67

Outline

date1.cpp (5 of 5)

```
      // last day of year
      else {
         ++year;
         month = 1;
         day = 1;
      }

} // end function helpIncrement

// overloaded output operator
ostream &operator<<( ostream &output, const Date &d )
{
   static char *monthName[ 13 ] = { "", "January",
      "February", "March", "April", "May", "June",
      "July", "August", "September", "October",
      "November", "December" };

   output << monthName[ d.month ] << ' '
          << d.day << ", " << d.year;

   return output;   // enables cascading

} // end function operator<<
```

Figure 3.25: **Date** class member-and **friend**-function definition. (part 3 of 3)



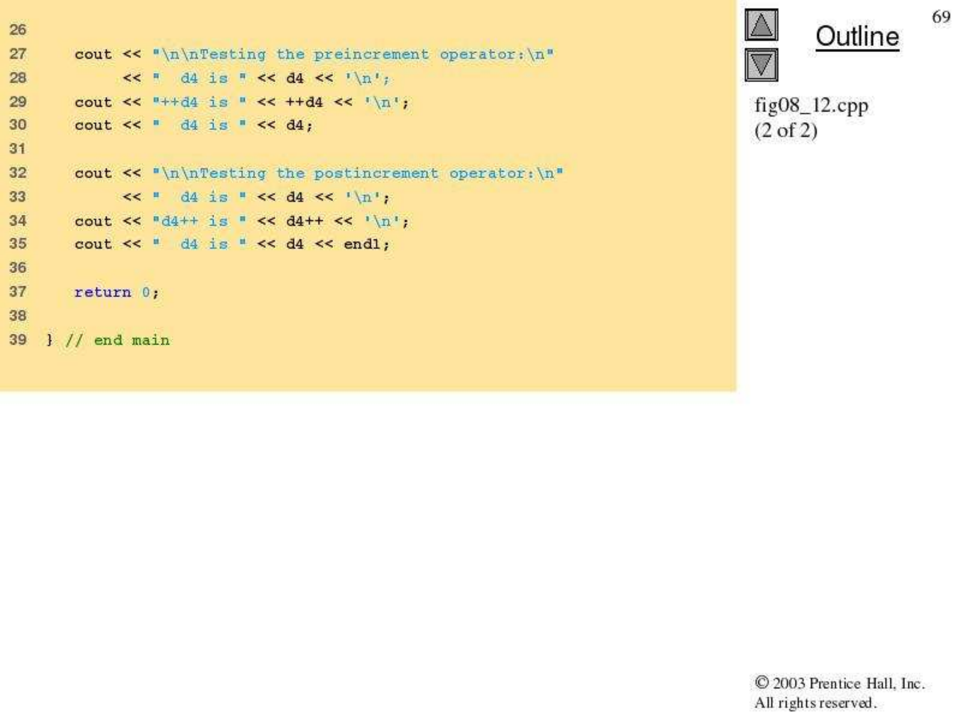
```
// Fig. 8.12: fig08_12.cpp
// Date class test program.
#include <iostream>

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

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

int main()
{
   Date d1;  // defaults to January 1, 1900
   Date d2( 12, 27, 1992 );
   Date d3( 0, 99, 8045 );  // invalid date

   cout << "d1 is " << d1 << "\nd2 is " << d2
        << "\nd3 is " << d3;

   cout << "\n\nd2 += 7 is " << ( d2 += 7 );

   d3.setDate( 2, 28, 1992 );
   cout << "\n\n  d3 is " << d3;
   cout << "\n++d3 is " << ++d3;

   Date d4( 7, 13, 2002 );
```

68

Outline

fig08_12.cpp
(1 of 2)

© 2003 Prentice Hall, Inc.
All rights reserved.



```

   cout << "\n\nTesting the preincrement operator:\n"
        << "  d4 is " << d4 << '\n';
   cout << "++d4 is " << ++d4 << '\n';
   cout << "  d4 is " << d4;

   cout << "\n\nTesting the postincrement operator:\n"
        << "  d4 is " << d4 << '\n';
   cout << "d4++ is " << d4++ << '\n';
   cout << "  d4 is " << d4 << endl;

   return 0;

} // end main
```

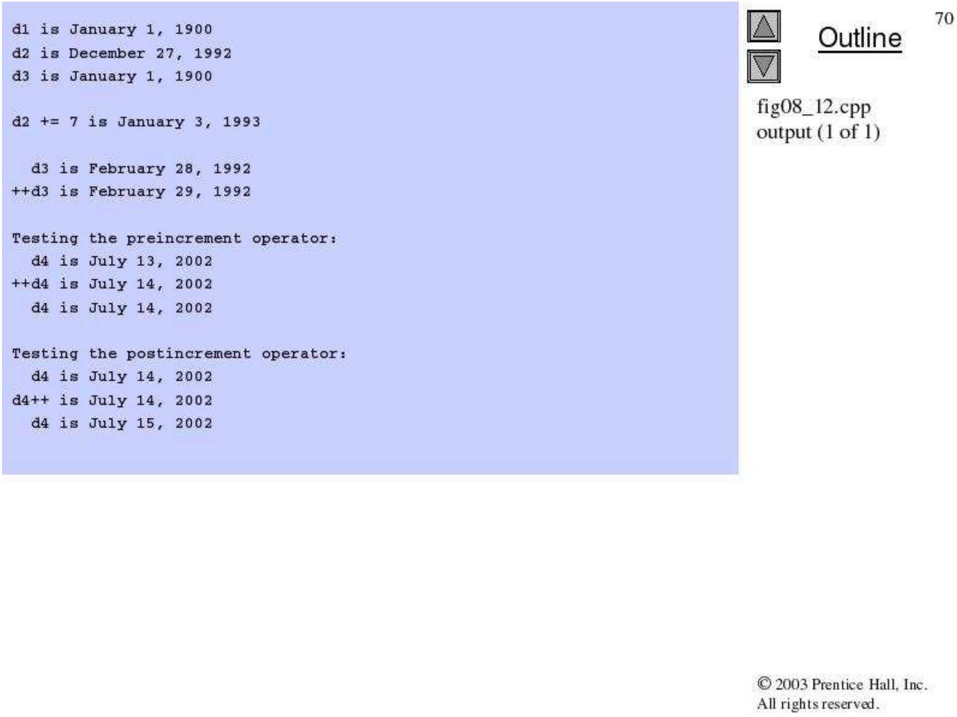
69

Outline

fig08_12.cpp
(2 of 2)

© 2003 Prentice Hall, Inc.
All rights reserved.

Figure 3.26: **Date** class test program.



```
d1 is January 1, 1900
d2 is December 27, 1992
d3 is January 1, 1900

d2 += 7 is January 3, 1993

  d3 is February 28, 1992
++d3 is February 29, 1992

Testing the preincrement operator:
  d4 is July 13, 2002
++d4 is July 14, 2002
  d4 is July 14, 2002

Testing the postincrement operator:
  d4 is July 14, 2002
d4++ is July 14, 2002
  d4 is July 15, 2002
```

fig08_12.cpp
output (1 of 1)

Figure 3.27: **Date** class test program, output.

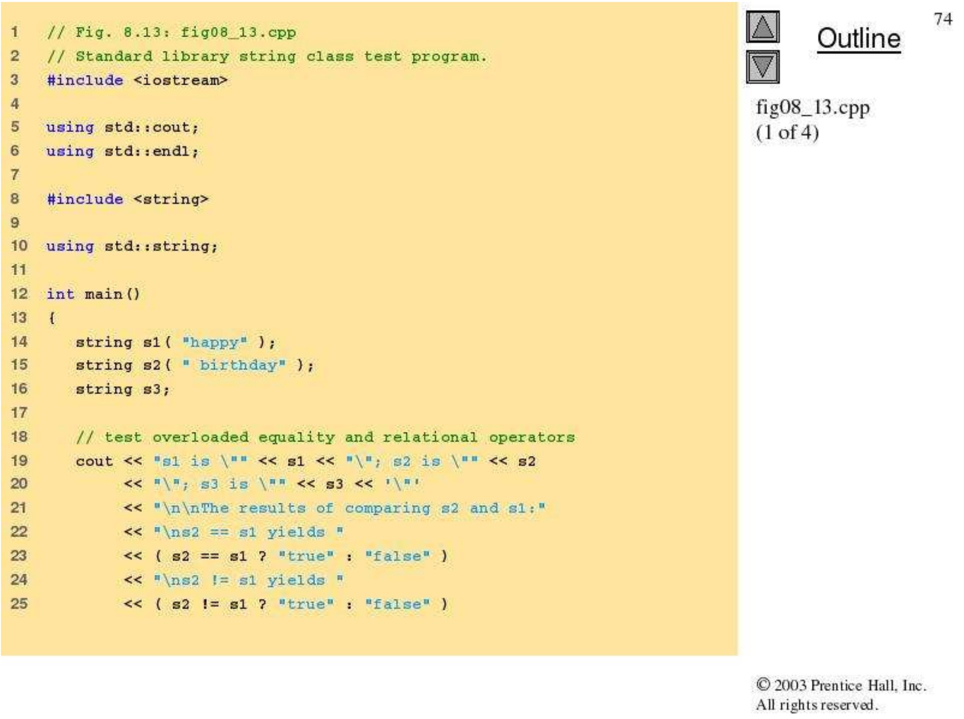
3.13 Standard Library Classes string and vector

We learned that we can build a **String** (**Array**) class that is better than the C-style, **char *** strings (pointer-based arrays) that C++ absorbed from C.

- Classes built into C++
  - Available for anyone to use
  - **string** ; Similar to our **String** class
  - **vector**; Dynamically resizable array
- Redo our **String** and **Array** examples
  - Use **string** and **vector**
- Class string
  - Header <**string**>, namespace **std**

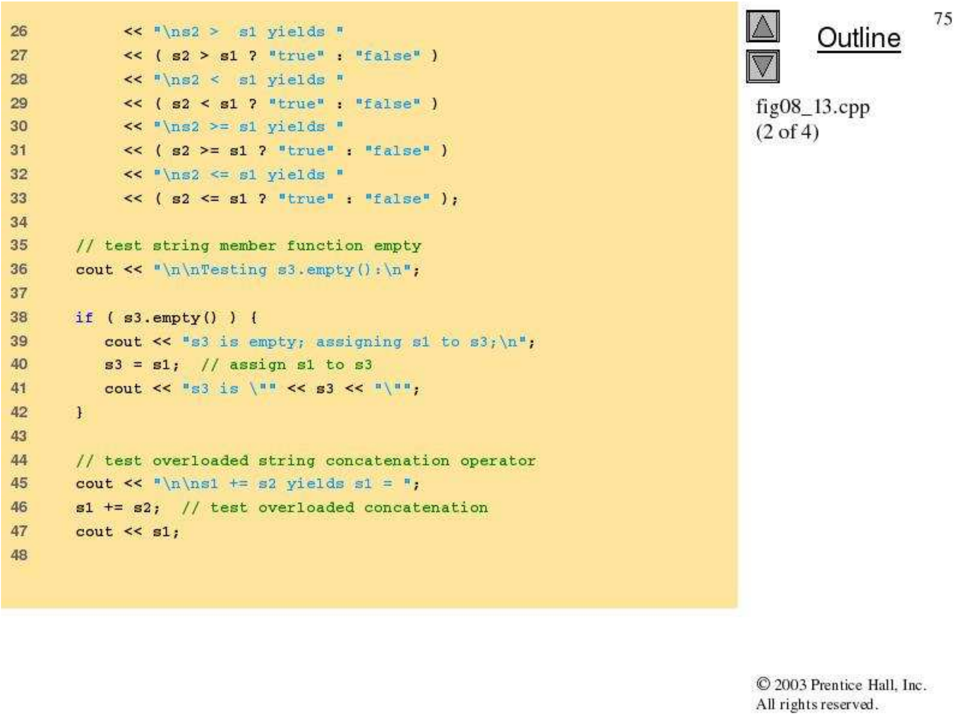
- Can initialize **string s1("hi");**
- Overloaded <<; **cout** << **s1**
- Overloaded relational operators; == **!=** >= > <= <
- Assignment operator =
- Concatenation (overloaded +=)
- Substring function **substr**
  - **s1.substr(0, 14);** ; Starts at location 0, gets 14 characters
  - **S1.substr(15)** ; Substring beginning at location 15
- Overloaded []
  - Access one character
  - No range checking (if subscript invalid)
- **at** function
  - **s1.at(10)**
  - Character at subscript 10
  - Has bounds checking; will end program if invalid (learn more in Chapter 13)

The programs of Figs. 3.28-3.30 reimplements the program of Figs. 3.18-3.21, using standart class **string**.



```
// Fig. 8.13: fig08_13.cpp
// Standard library string class test program.
#include <iostream>

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

#include <string>

using std::string;

int main()
{
   string s1( "happy" );
   string s2( " birthday" );
   string s3;

   // test overloaded equality and relational operators
   cout << "s1 is \"" << s1 << "\"; s2 is \"" << s2
        << "\"; s3 is \"" << s3 << '\"'
        << "\n\nThe results of comparing s2 and s1:"
        << "\ns2 == s1 yields "
        << ( s2 == s1 ? "true" : "false" )
        << "\ns2 != s1 yields "
        << ( s2 != s1 ? "true" : "false" )
```

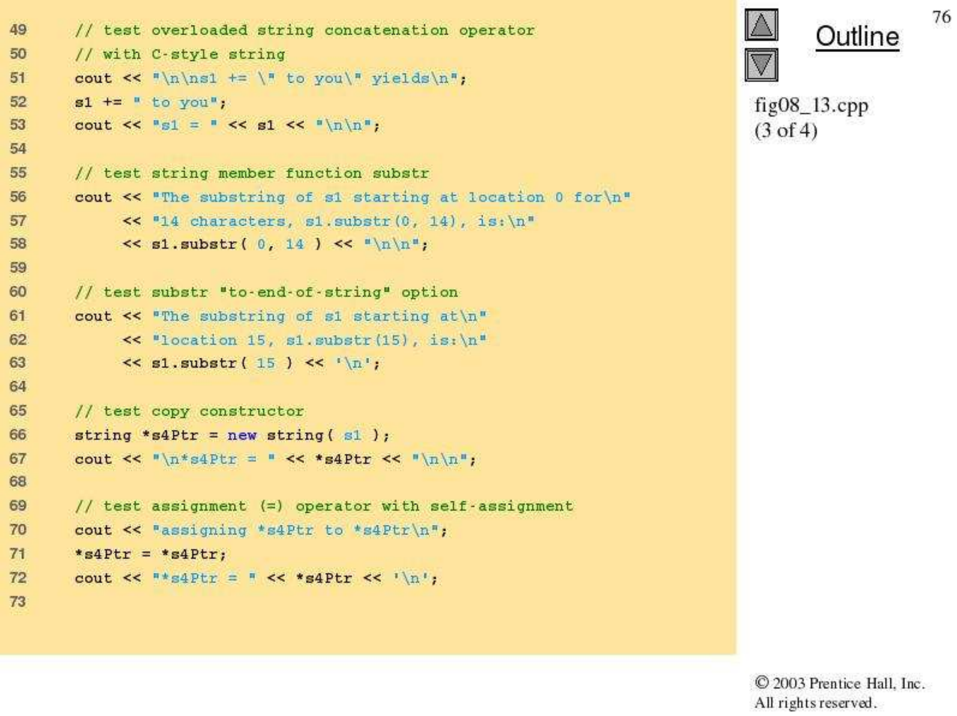
© 2003 Prentice Hall, Inc.
All rights reserved.



```
        << "\ns2 >  s1 yields "
        << ( s2 > s1 ? "true" : "false" )
        << "\ns2 <  s1 yields "
        << ( s2 < s1 ? "true" : "false" )
        << "\ns2 >= s1 yields "
        << ( s2 >= s1 ? "true" : "false" )
        << "\ns2 <= s1 yields "
        << ( s2 <= s1 ? "true" : "false" );

   // test string member function empty
   cout << "\n\nTesting s3.empty():\n";

   if ( s3.empty() ) {
      cout << "s3 is empty; assigning s1 to s3;\n";
      s3 = s1;  // assign s1 to s3
      cout << "s3 is \"" << s3 << "\"";
   }

   // test overloaded string concatenation operator
   cout << "\n\ns1 += s2 yields s1 = ";
   s1 += s2;  // test overloaded concatenation
   cout << s1;

```

© 2003 Prentice Hall, Inc.
All rights reserved.

Figure 3.28: Standart library class **string** (part 1 of 2).



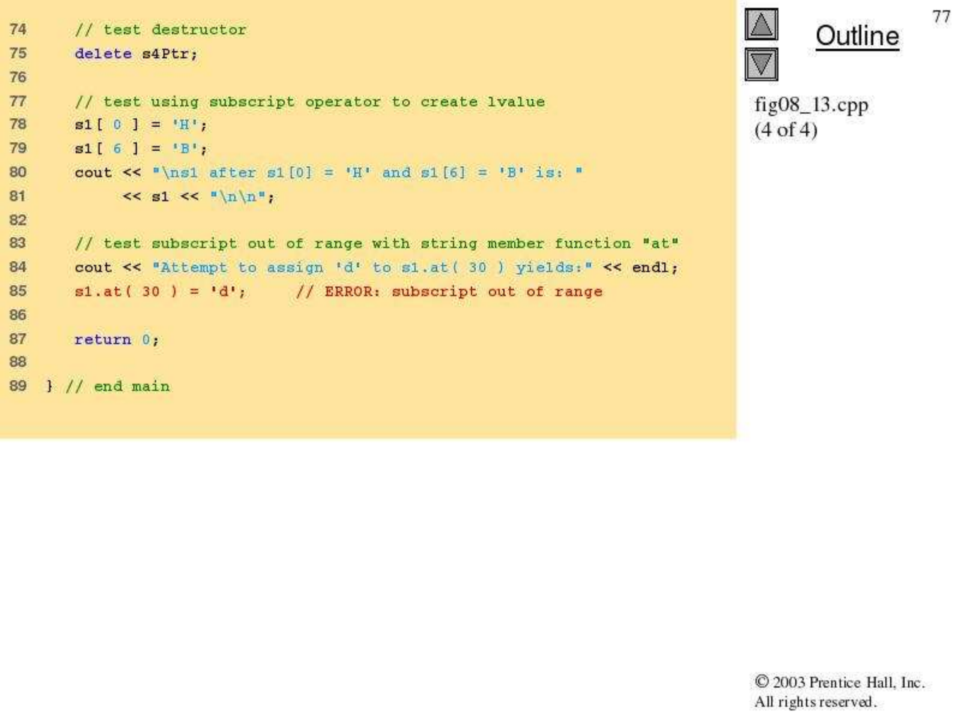
```
    // test overloaded string concatenation operator
    // with C-style string
    cout << "\n\ns1 += \" to you\" yields\n";
    s1 += " to you";
    cout << "s1 = " << s1 << "\n\n";

    // test string member function substr
    cout << "The substring of s1 starting at location 0 for\n"
         << "14 characters, s1.substr(0, 14), is:\n"
         << s1.substr( 0, 14 ) << "\n\n";

    // test substr "to-end-of-string" option
    cout << "The substring of s1 starting at\n"
         << "location 15, s1.substr(15), is:\n"
         << s1.substr( 15 ) << '\n';

    // test copy constructor
    string *s4Ptr = new string( s1 );
    cout << "\n*s4Ptr = " << *s4Ptr << "\n\n";

    // test assignment (=) operator with self-assignment
    cout << "assigning *s4Ptr to *s4Ptr\n";
    *s4Ptr = *s4Ptr;
    cout << "*s4Ptr = " << *s4Ptr << '\n';

```

76

Outline

fig08_13.cpp
(3 of 4)

© 2003 Prentice Hall, Inc.
All rights reserved.



```
    // test destructor
    delete s4Ptr;

    // test using subscript operator to create lvalue
    s1[ 0 ] = 'H';
    s1[ 6 ] = 'B';
    cout << "\ns1 after s1[0] = 'H' and s1[6] = 'B' is: "
         << s1 << "\n\n";

    // test subscript out of range with string member function "at"
    cout << "Attempt to assign 'd' to s1.at( 30 ) yields:" << endl;
    s1.at( 30 ) = 'd';      // ERROR: subscript out of range

    return 0;

} // end main
```

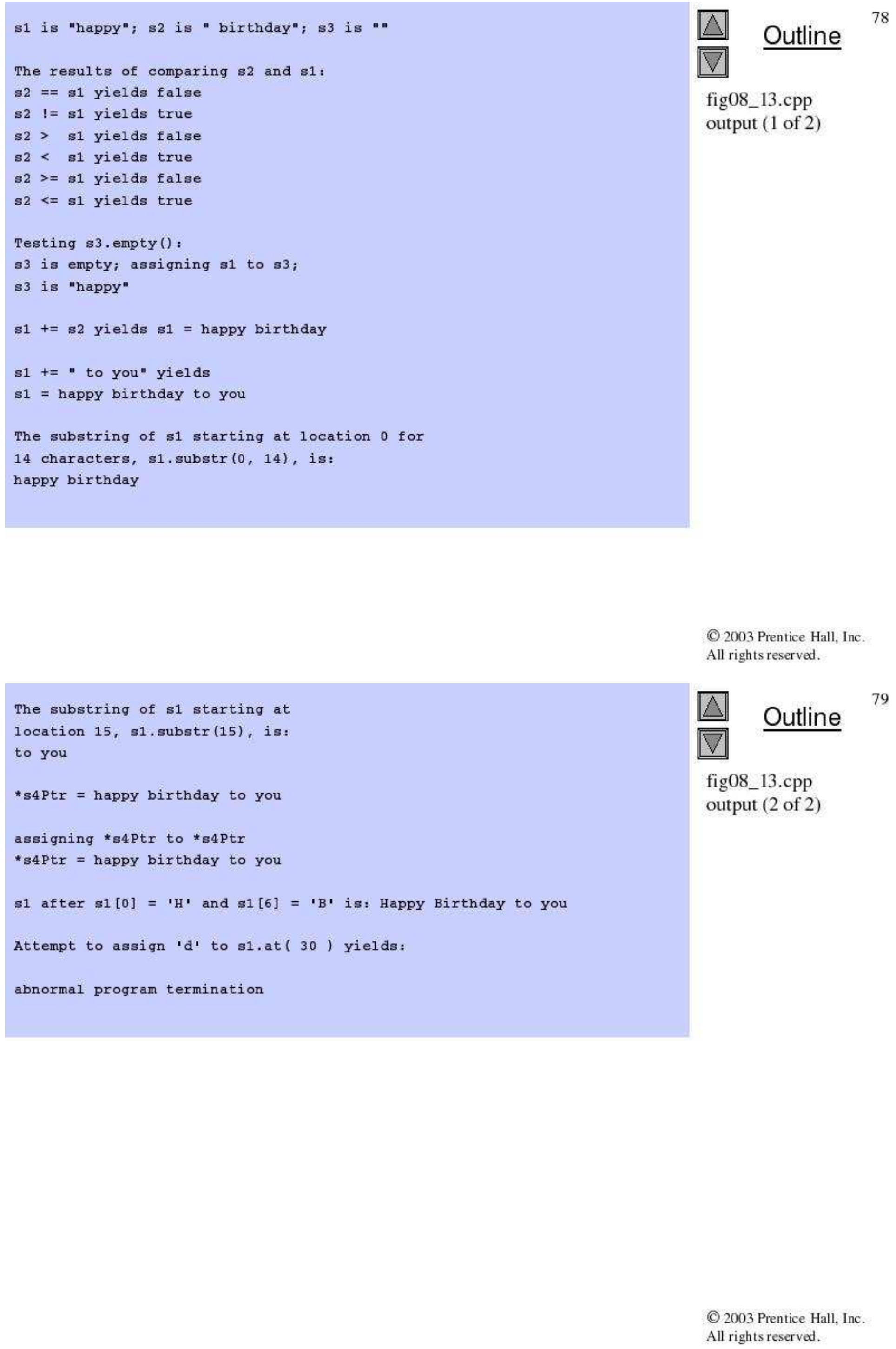
77

Outline

fig08_13.cpp
(4 of 4)

© 2003 Prentice Hall, Inc.
All rights reserved.

Figure 3.29: Standart library class **string** (part 2 of 2).



```
s1 is "happy"; s2 is " birthday"; s3 is ""

The results of comparing s2 and s1:
s2 == s1 yields false
s2 != s1 yields true
s2 >  s1 yields false
s2 <  s1 yields true
s2 >= s1 yields false
s2 <= s1 yields true

Testing s3.empty():
s3 is empty; assigning s1 to s3;
s3 is "happy"

s1 += s2 yields s1 = happy birthday

s1 += " to you" yields
s1 = happy birthday to you

The substring of s1 starting at location 0 for
14 characters, s1.substr(0, 14), is:
happy birthday
```

fig08_13.cpp
output (1 of 2)

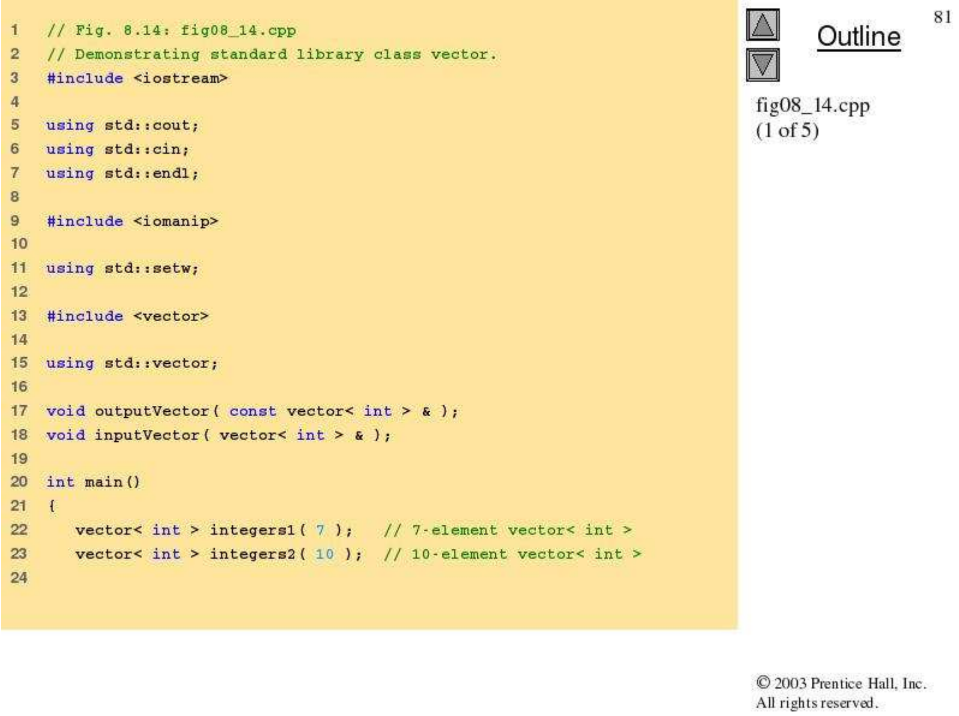
```
The substring of s1 starting at
location 15, s1.substr(15), is:
to you

*s4Ptr = happy birthday to you

assigning *s4Ptr to *s4Ptr
*s4Ptr = happy birthday to you

s1 after s1[0] = 'H' and s1[6] = 'B' is: Happy Birthday to you

Attempt to assign 'd' to s1.at( 30 ) yields:

abnormal program termination
```

fig08_13.cpp
output (2 of 2)

Figure 3.30: Standart library class **string**, output.

- Class vector
  - Header <**vector**>, namespace **std**
  - Store any type; **vector**< **int** > **myArray(10)**
  - Function **size** (**myArray.size()**)
  - Overloaded []; get specific element, **myArray**[**3**]
  - Overloaded **!=**, **==**, and **=**; inequality, equality, assignment

The programs of Figs. 3.31-3.34 reimplements the program of Figs. 3.8-3.11, using standart class **vector**.



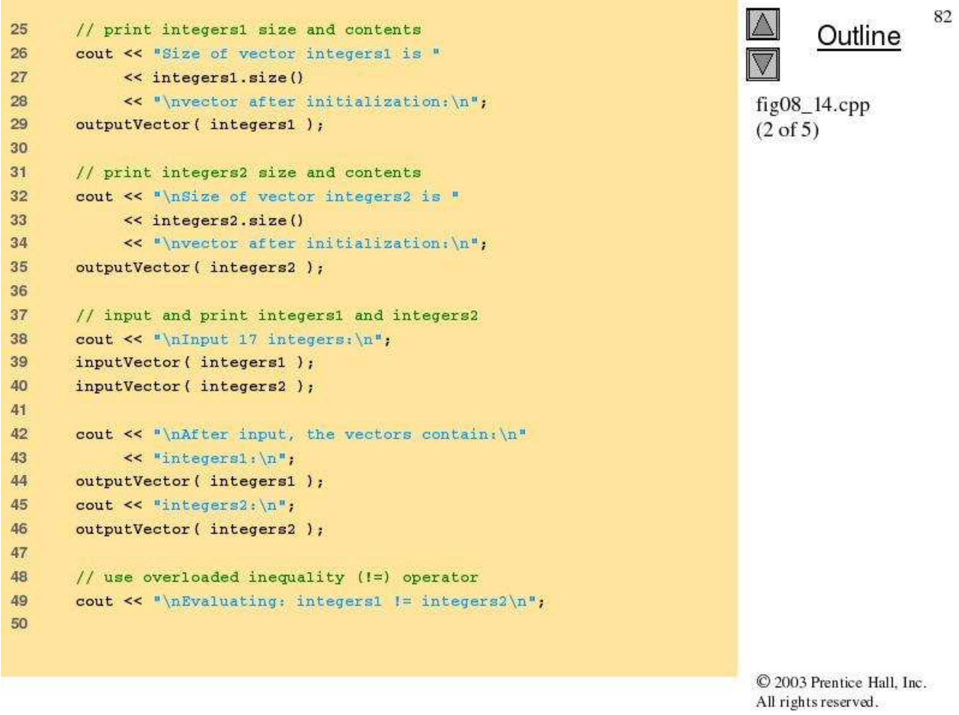
```cpp
// Fig. 8.14: fig08_14.cpp
// Demonstrating standard library class vector.
#include <iostream>

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

#include <iomanip>

using std::setw;

#include <vector>

using std::vector;

void outputVector( const vector< int > & );
void inputVector( vector< int > & );

int main()
{
   vector< int > integers1( 7 );   // 7-element vector< int >
   vector< int > integers2( 10 );  // 10-element vector< int >

```

81

Outline

fig08_14.cpp
(1 of 5)

© 2003 Prentice Hall, Inc.
All rights reserved.



```cpp
   // print integers1 size and contents
   cout << "Size of vector integers1 is "
        << integers1.size()
        << "\nvector after initialization:\n";
   outputVector( integers1 );

   // print integers2 size and contents
   cout << "\nSize of vector integers2 is "
        << integers2.size()
        << "\nvector after initialization:\n";
   outputVector( integers2 );

   // input and print integers1 and integers2
   cout << "\nInput 17 integers:\n";
   inputVector( integers1 );
   inputVector( integers2 );

   cout << "\nAfter input, the vectors contain:\n"
        << "integers1:\n";
   outputVector( integers1 );
   cout << "integers2:\n";
   outputVector( integers2 );

   // use overloaded inequality (!=) operator
   cout << "\nEvaluating: integers1 != integers2\n";

```

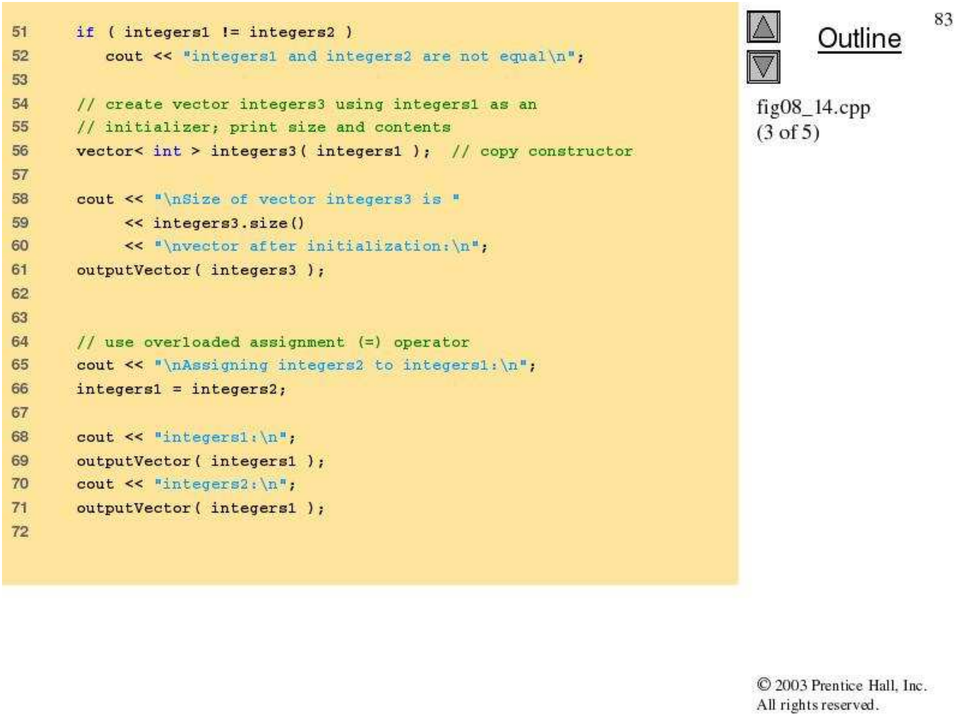
82

Outline

fig08_14.cpp
(2 of 5)

© 2003 Prentice Hall, Inc.
All rights reserved.

Figure 3.31: Standart library class **vector**. (part 1 of 3)

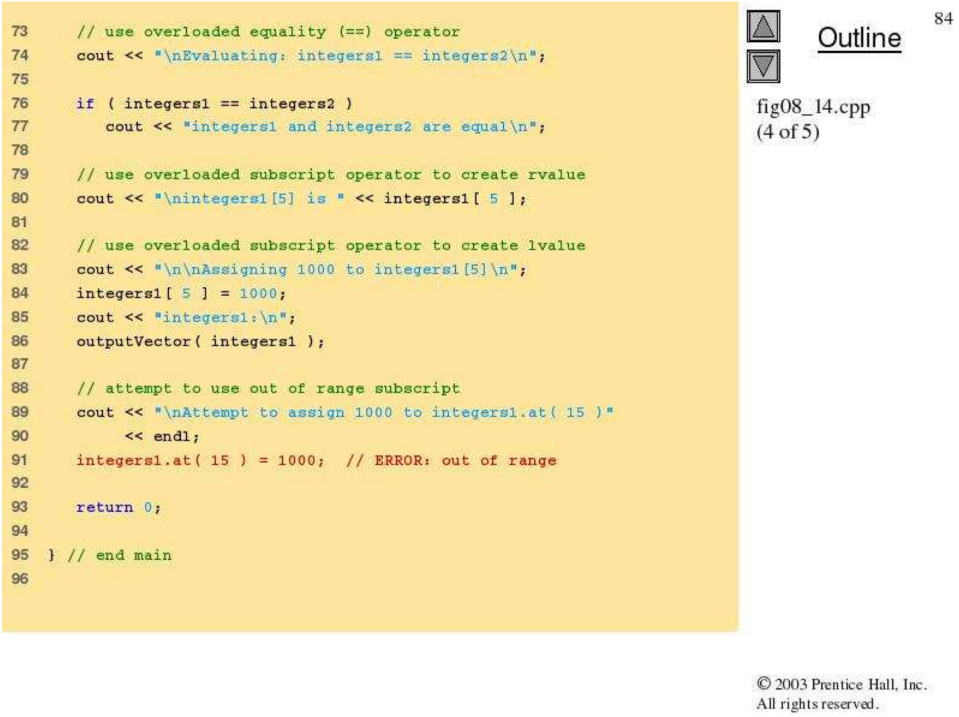


```
   if ( integers1 != integers2 )
      cout << "integers1 and integers2 are not equal\n";

   // create vector integers3 using integers1 as an
   // initializer; print size and contents
   vector< int > integers3( integers1 );  // copy constructor

   cout << "\nSize of vector integers3 is "
        << integers3.size()
        << "\nvector after initialization:\n";
   outputVector( integers3 );


   // use overloaded assignment (=) operator
   cout << "\nAssigning integers2 to integers1:\n";
   integers1 = integers2;

   cout << "integers1:\n";
   outputVector( integers1 );
   cout << "integers2:\n";
   outputVector( integers1 );

```

83

Outline

fig08_14.cpp
(3 of 5)



```
   // use overloaded equality (==) operator
   cout << "\nEvaluating: integers1 == integers2\n";

   if ( integers1 == integers2 )
      cout << "integers1 and integers2 are equal\n";

   // use overloaded subscript operator to create rvalue
   cout << "\nintegers1[5] is " << integers1[ 5 ];

   // use overloaded subscript operator to create lvalue
   cout << "\n\nAssigning 1000 to integers1[5]\n";
   integers1[ 5 ] = 1000;
   cout << "integers1:\n";
   outputVector( integers1 );

   // attempt to use out of range subscript
   cout << "\nAttempt to assign 1000 to integers1.at( 15 )"
        << endl;
   integers1.at( 15 ) = 1000;  // ERROR: out of range

   return 0;

} // end main

```

84

Outline

fig08_14.cpp
(4 of 5)

Figure 3.32: Standart library class **vector**. (part 2 of 3)

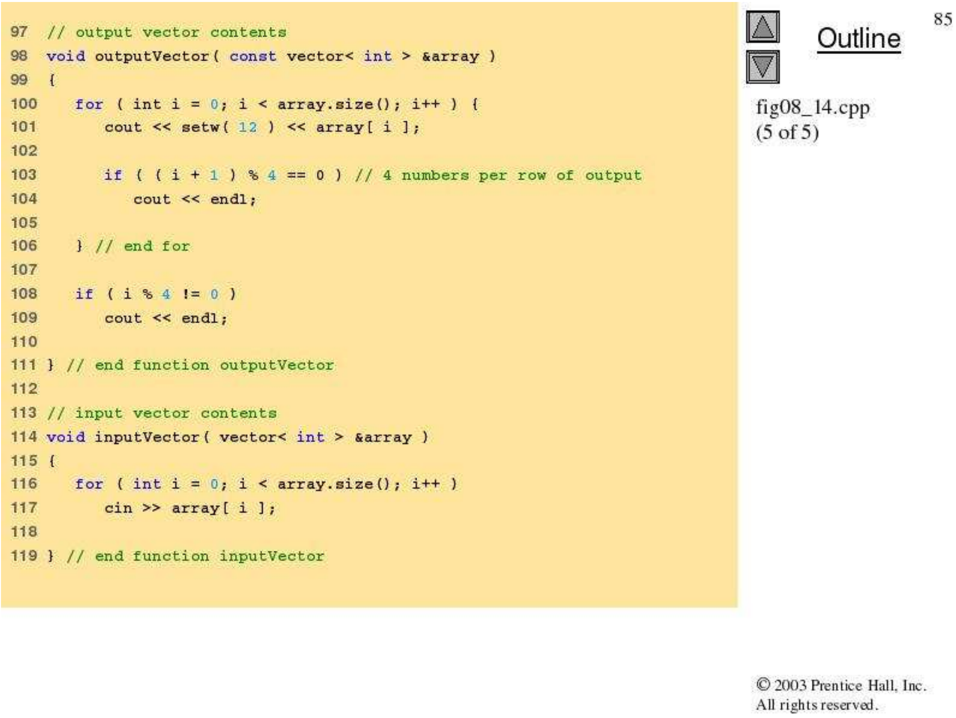
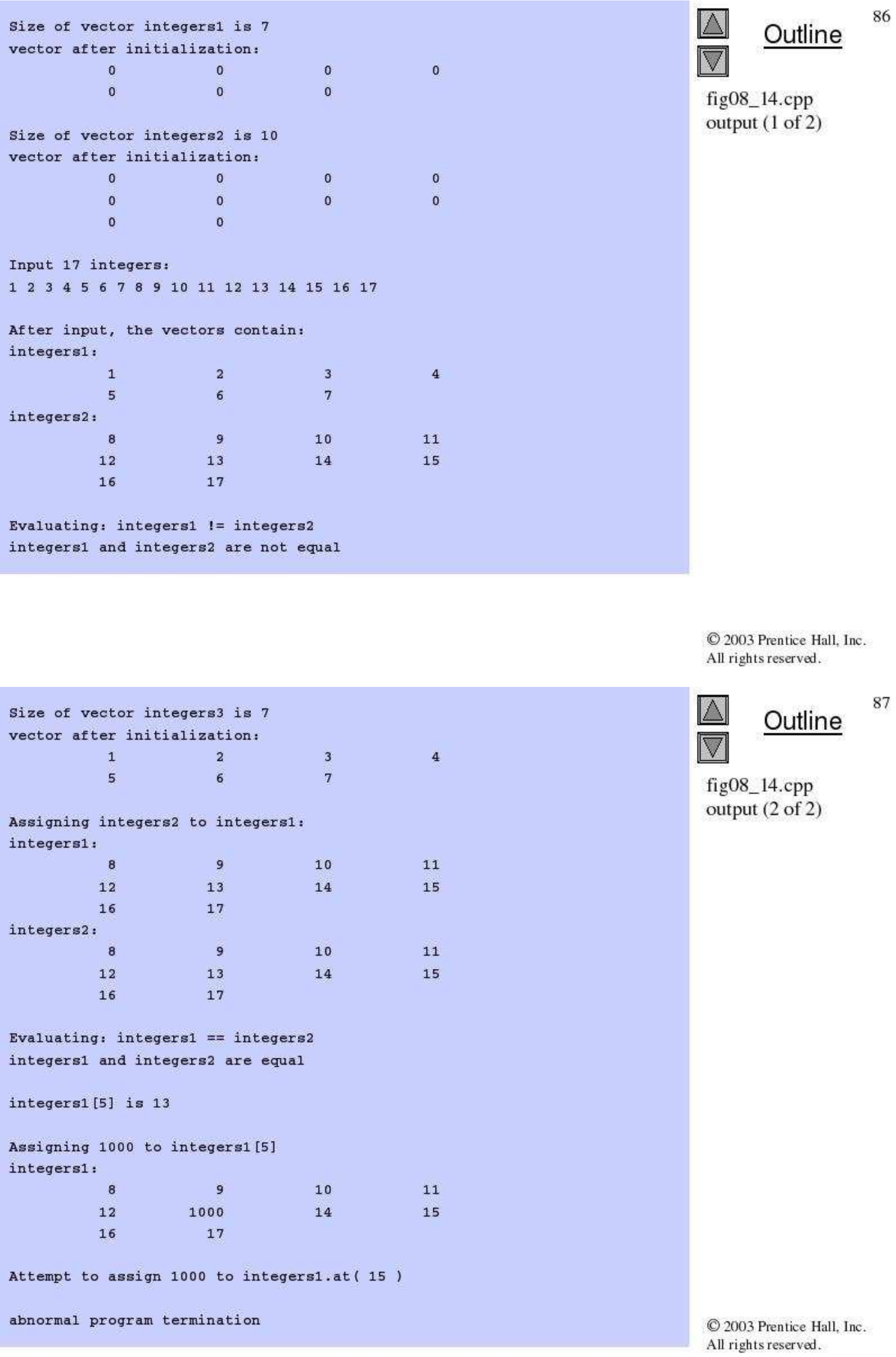


fig08_14.cpp
(5 of 5)

```
// output vector contents
void outputVector( const vector< int > &array )
{
   for ( int i = 0; i < array.size(); i++ ) {
      cout << setw( 12 ) << array[ i ];

      if ( ( i + 1 ) % 4 == 0 ) // 4 numbers per row of output
         cout << endl;

   } // end for

   if ( i % 4 != 0 )
      cout << endl;

} // end function outputVector

// input vector contents
void inputVector( vector< int > &array )
{
   for ( int i = 0; i < array.size(); i++ )
      cin >> array[ i ];

} // end function inputVector
```

Figure 3.33: Standart library class **vector**. (part 3 of 3)



86

Outline

fig08_14.cpp
output (1 of 2)

```
Size of vector integers1 is 7
vector after initialization:
           0           0           0           0
           0           0           0

Size of vector integers2 is 10
vector after initialization:
           0           0           0           0
           0           0           0           0
           0           0

Input 17 integers:
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17

After input, the vectors contain:
integers1:
           1           2           3           4
           5           6           7
integers2:
           8           9          10          11
          12          13          14          15
          16          17

Evaluating: integers1 != integers2
integers1 and integers2 are not equal
```

87

Outline

fig08_14.cpp
output (2 of 2)

```
Size of vector integers3 is 7
vector after initialization:
           1           2           3           4
           5           6           7

Assigning integers2 to integers1:
integers1:
           8           9          10          11
          12          13          14          15
          16          17
integers2:
           8           9          10          11
          12          13          14          15
          16          17

Evaluating: integers1 == integers2
integers1 and integers2 are equal

integers1[5] is 13

Assigning 1000 to integers1[5]
integers1:
           8           9          10          11
          12        1000          14          15
          16          17

Attempt to assign 1000 to integers1.at( 15 )

abnormal program termination
```

Figure 3.34: Standart library class **vector**, output.

Chapter 4

Object-Oriented Programming: Inheritance

4.1 Introduction

- Inheritance
    - Software reusability
    - Create new class from existing class
        - Absorb existing class's data and behaviors
        - Enhance with new capabilities
    - Derived class inherits from base class
        - Derived class
            - More specialized group of objects
            - Behaviors inherited from base class; can customize
            - Additional behaviors
- Class hierarchy
    - Direct base class; inherited explicitly (one level up hierarchy)
    - Indirect base class; inherited two or more levels up hierarchy
    - Single inheritance; inherits from one base class
    - Multiple inheritance; Inherits from multiple base classes (Base classes possibly unrelated); Chapter 22
- Three types of inheritance
    - **public**

        - Every object of derived class also object of base class
            - Base-class objects not objects of derived classes
            - Example: All cars vehicles, but not all vehicles cars
        - Can access non-**private** members of base class
            - Derived class can effect change to **private** base-class members
            - Through inherited non-**private** member functions
    - **private**
        - Alternative to composition
        - Chapter 17
    - **protected**
        - Rarely used

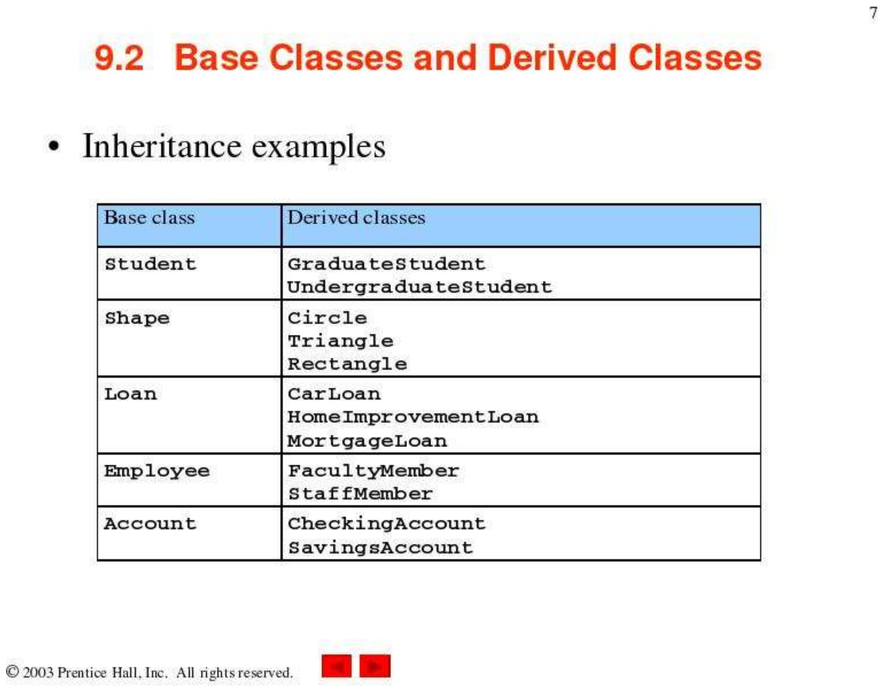
- Abstraction
    - Focus on commonalities among objects in system; "is-a" vs. "has-a"
    - "is-a"
        - Inheritance
        - Derived class object treated as base class object
        - Example: Car *is a* vehicle; Vehicle properties/behaviors also car properties/behaviors
    - "has-a"
        - Composition
        - Object contains one or more objects of other classes as members
        - Example: Car *has a* steering wheel

4.2 Base Classes and Derived Classes

- Base classes and derived classes
    - Object of one class "is an" object of another class
        - Example: Rectangle is quadrilateral.
            - Class **Rectangle** inherits from class **Quadrilateral**
            - **Quadrilateral**: base class

      - **Rectangle**: derived class
  - Base class typically represents larger set of objects than derived classes
    - Example:
      - Base class: **Vehicle**
        Cars, trucks, boats, bicycles, . . .
      - Derived class: **Car**
        Smaller, more-specific subset of vehicles

- Inheritance examples (see Fig. 4.1)



7

9.2 Base Classes and Derived Classes

- Inheritance examples

Base class	Derived classes
`Student`	`GraduateStudent` `UndergraduateStudent`

`Shape`	`Circle`
	`Triangle`
	`Rectangle`

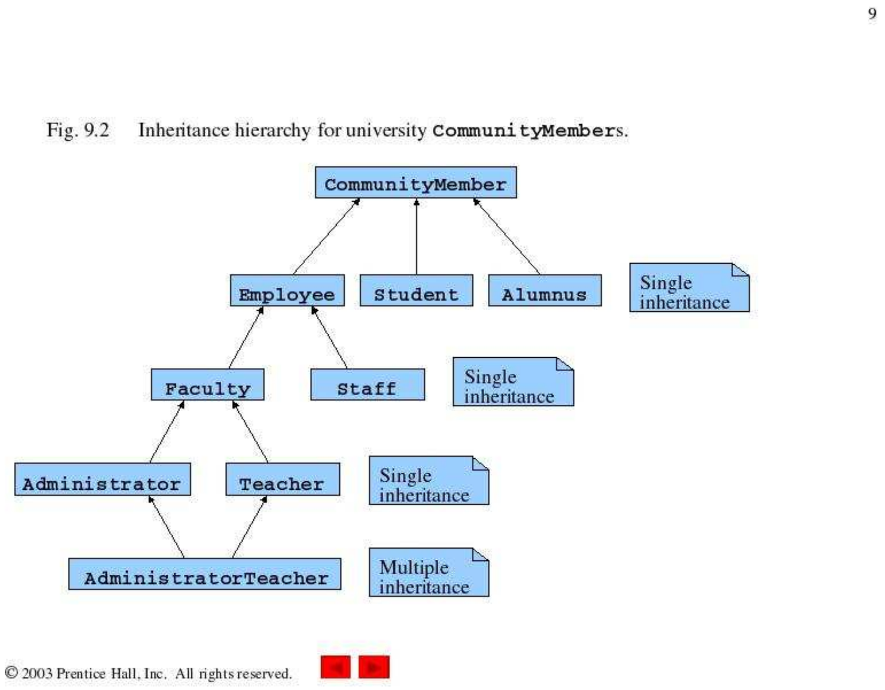
`Loan`	`CarLoan`
	`HomeImprovementLoan`
	`MortgageLoan`

`Employee`	`FacultyMember` `StaffMember`
`Account`	`CheckingAccount` `SavingsAccount`

© 2003 Prentice Hall, Inc. All rights reserved.

Figure 4.1: Inheritance examples

- Inheritance hierarchy (see Fig. 4.2 Top)
  - Inheritance relationships: tree-like hierarchy structure
  - Each class becomes
    - Base class; supply data/behaviors to other classes
    - OR



9

Fig. 9.2 Inheritance hierarchy for university CommunityMembers.

CommunityMember

Employee

Student

Alumnus

Single inheritance

Faculty

Staff

Single inheritance

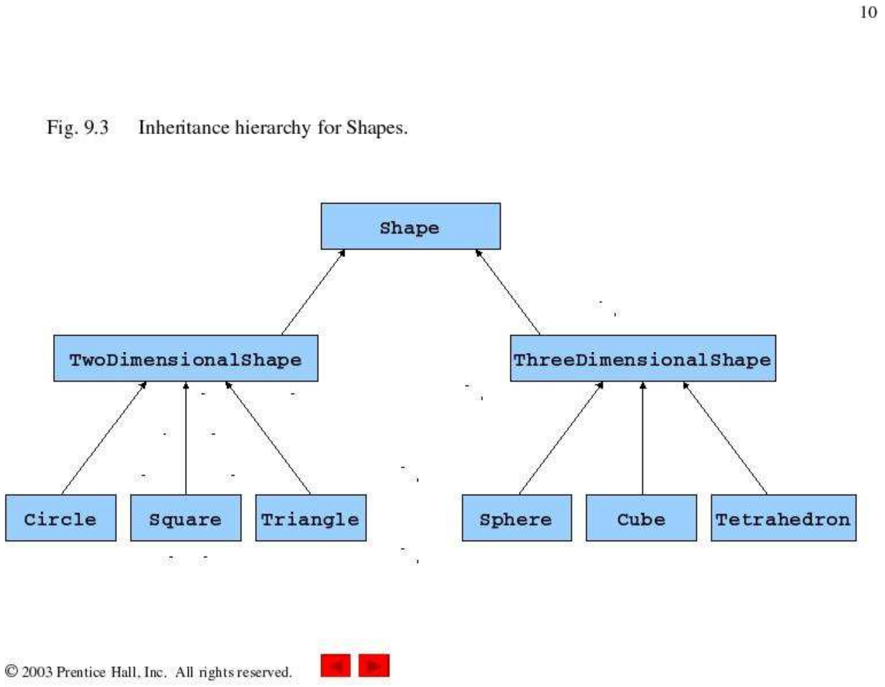
Administrator

Teacher

Single inheritance

AdministratorTeacher

Multiple inheritance



10

Fig. 9.3 Inheritance hierarchy for Shapes.

Shape

TwoDimensionalShape

ThreeDimensionalShape

Circle

Square

Triangle

Sphere

Cube

Tetrahedron

Figure 4.2: Inheritance hierarchy for university **CommunityMembers** and Inheritance hierarchy for **Shape**s

    - * Derived class; inherit data/behaviors from other classes

- public inheritance
  - Specify with:
  - **Class TwoDimensionalShape : public Shape**
    Class **TwoDimensionalShape** inherits from class **Shape** (see Fig. 4.2 Bottom)
  - Base class **private** members
    - * Not accessible directly
    - * Still inherited; manipulate through inherited member functions
  - Base class **public** and **protected** members; inherited with original member access
  - **friend** functions; not inherited

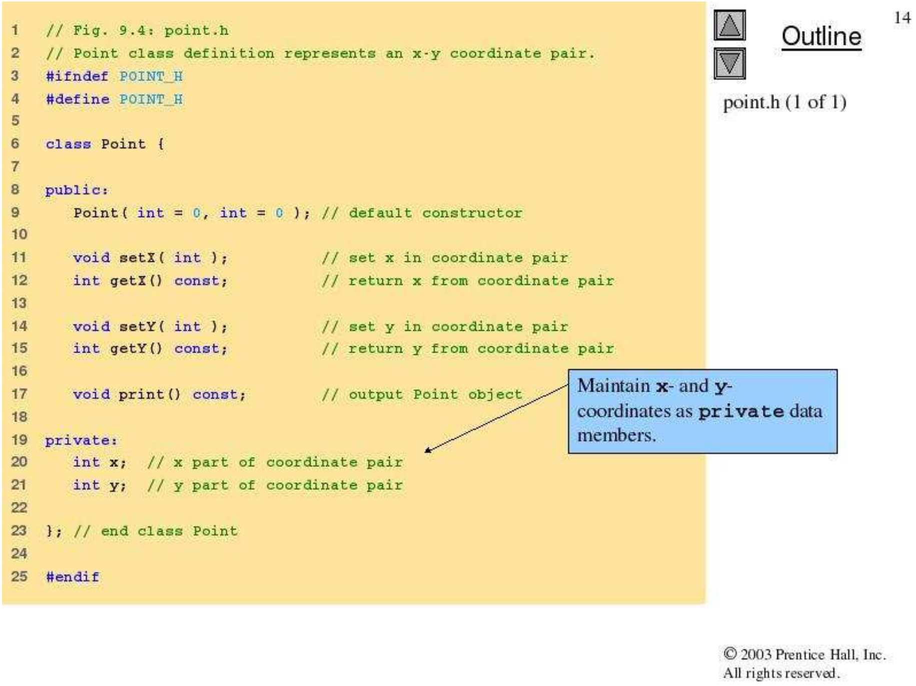
4.3 protected Members

Protected access

- Intermediate level of protection between **public** and **private**
- **protected** members accessible to
  - Base class members
  - Base class **friend**s
  - Derived class members
  - Derived class **friend**s
- Derived-class members
  - Refer to **public** and **protected** members of base class; simply use member names

4.4 Relationship between *Base Classes* and *Derived Classes*

- Base class and derived class relationship



14

Outline

point.h (1 of 1)

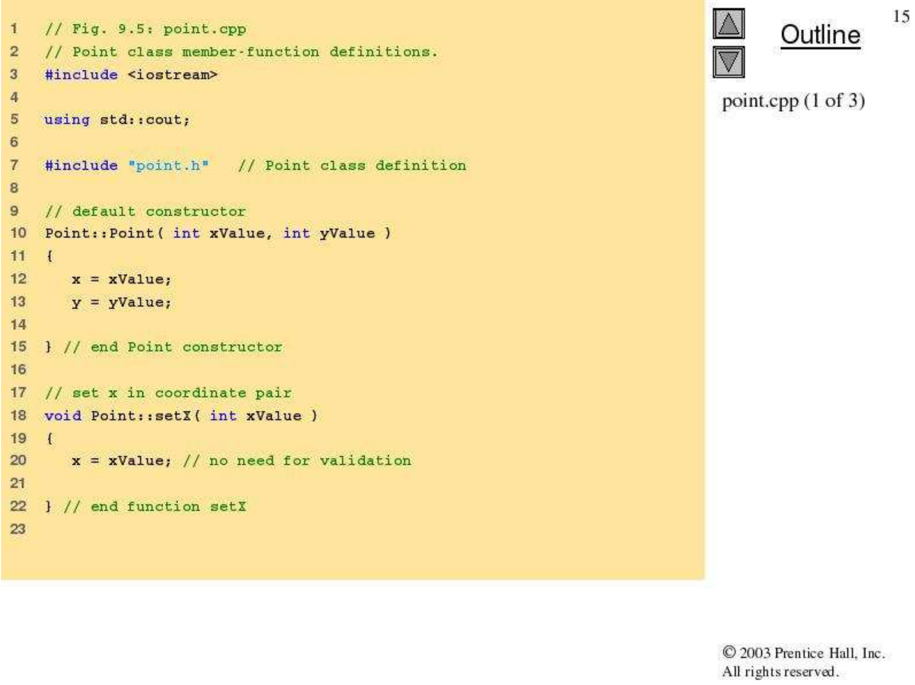
```
// Fig. 9.4: point.h
// Point class definition represents an x-y coordinate pair.
#ifndef POINT_H
#define POINT_H

class Point {

public:
   Point( int = 0, int = 0 ); // default constructor

   void setX( int );          // set x in coordinate pair
   int getX() const;          // return x from coordinate pair

   void setY( int );          // set y in coordinate pair
   int getY() const;          // return y from coordinate pair

   void print() const;        // output Point object

private:
   int x;  // x part of coordinate pair
   int y;  // y part of coordinate pair

}; // end class Point

#endif
```

Maintain **x**- and **y**-coordinates as **private** data members.

Figure 4.3: **Point** class header file

- Example: Point/circle inheritance hierarchy
  - Point
    x-y coordinate pair
  - Circle
    x-y coordinate pair
    Radius

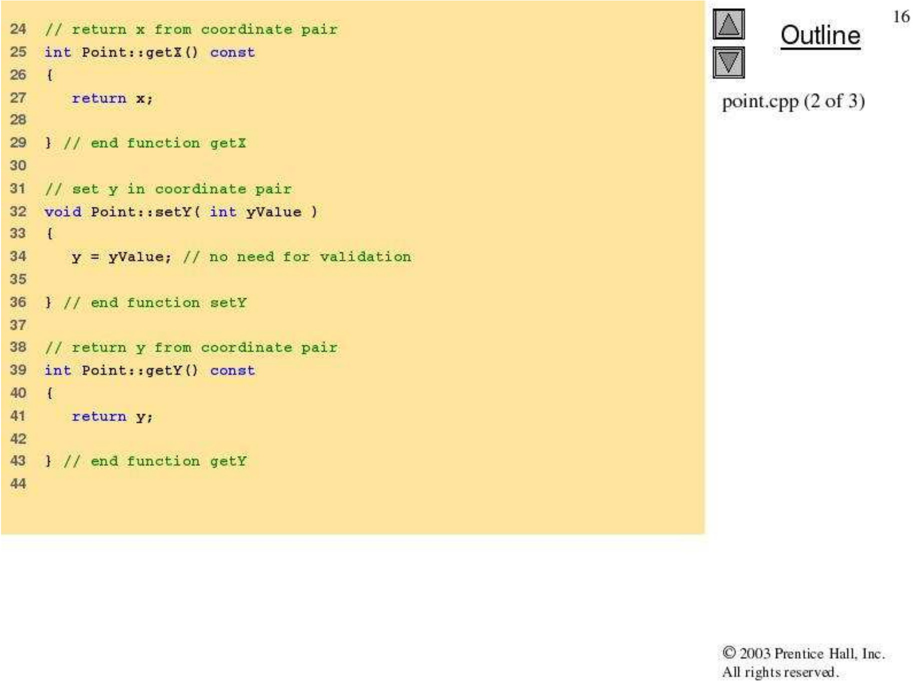
- Using protected data members
  - Advantages



```
// Fig. 9.5: point.cpp
// Point class member-function definitions.
#include <iostream>

using std::cout;

#include "point.h"   // Point class definition

// default constructor
Point::Point( int xValue, int yValue )
{
   x = xValue;
   y = yValue;

} // end Point constructor

// set x in coordinate pair
void Point::setX( int xValue )
{
   x = xValue; // no need for validation

} // end function setX

```

point.cpp (1 of 3)

© 2003 Prentice Hall, Inc.
All rights reserved.

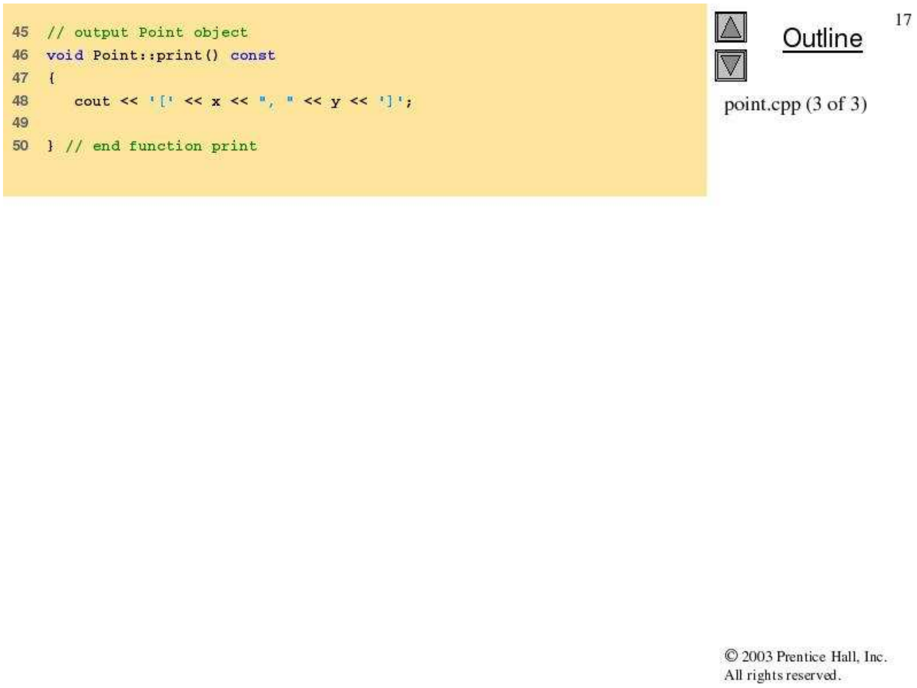


```
// return x from coordinate pair
int Point::getX() const
{
   return x;

} // end function getX

// set y in coordinate pair
void Point::setY( int yValue )
{
   y = yValue; // no need for validation

} // end function setY

// return y from coordinate pair
int Point::getY() const
{
   return y;

} // end function getY

```

point.cpp (2 of 3)

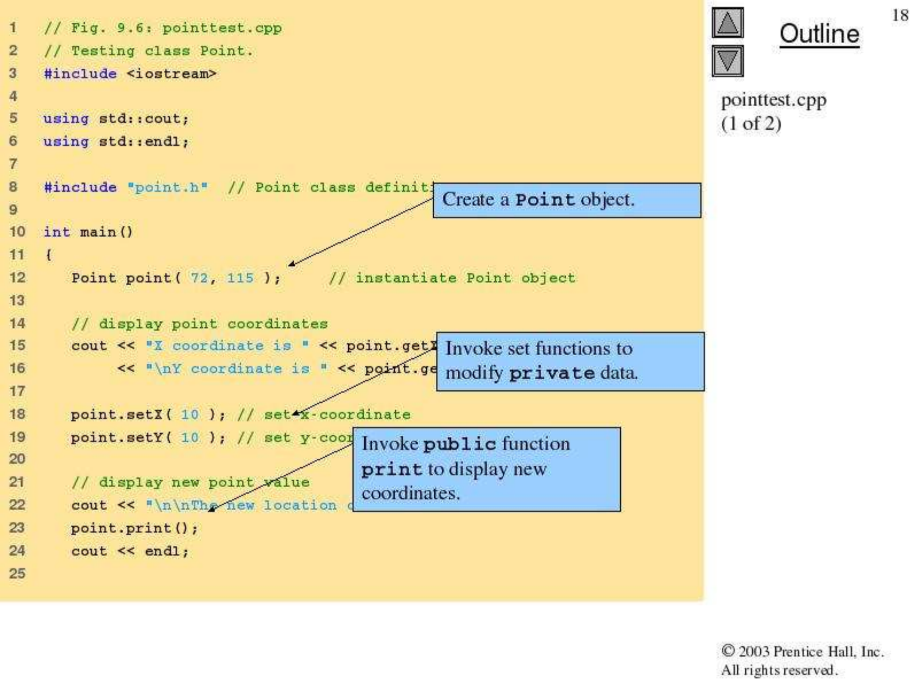
© 2003 Prentice Hall, Inc.
All rights reserved.

Figure 4.4: **Point** class represents an xy-coordinate pair. (part 1 of 2)



```
// output Point object
void Point::print() const
{
   cout << '[' << x << ", " << y << ']';

} // end function print
```

Figure 4.5: **Point** class represents an xy-coordinate pair. (part 2 of 2)



```
// Fig. 9.6: pointtest.cpp
// Testing class Point.
#include <iostream>

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

#include "point.h"  // Point class definit

int main()
{
   Point point( 72, 115 );     // instantiate Point object

   // display point coordinates
   cout << "X coordinate is " << point.getX
        << "\nY coordinate is " << point.ge

   point.setX( 10 ); // set x-coordinate
   point.setY( 10 ); // set y-coor

   // display new point value
   cout << "\n\nThe new location o
   point.print();
   cout << endl;

```

Create a **Point** object.

Invoke set functions to modify **private** data.

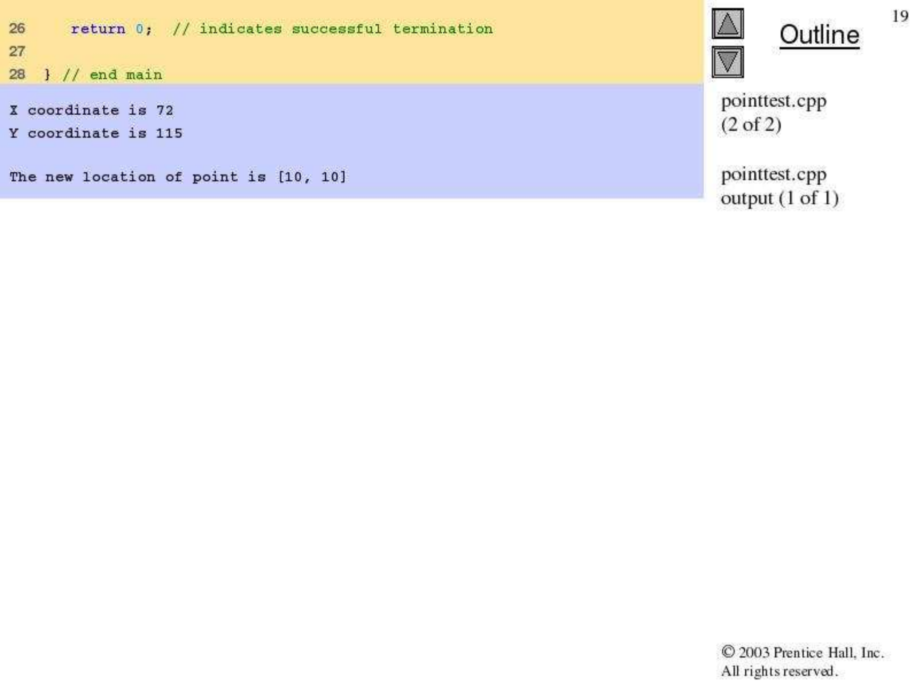
Invoke **public** function **print** to display new coordinates.

Outline

18

pointtest.cpp
(1 of 2)

© 2003 Prentice Hall, Inc.
All rights reserved.



```
   return 0;  // indicates successful termination

} // end main
```

```
X coordinate is 72
Y coordinate is 115

The new location of point is [10, 10]
```

Outline

19

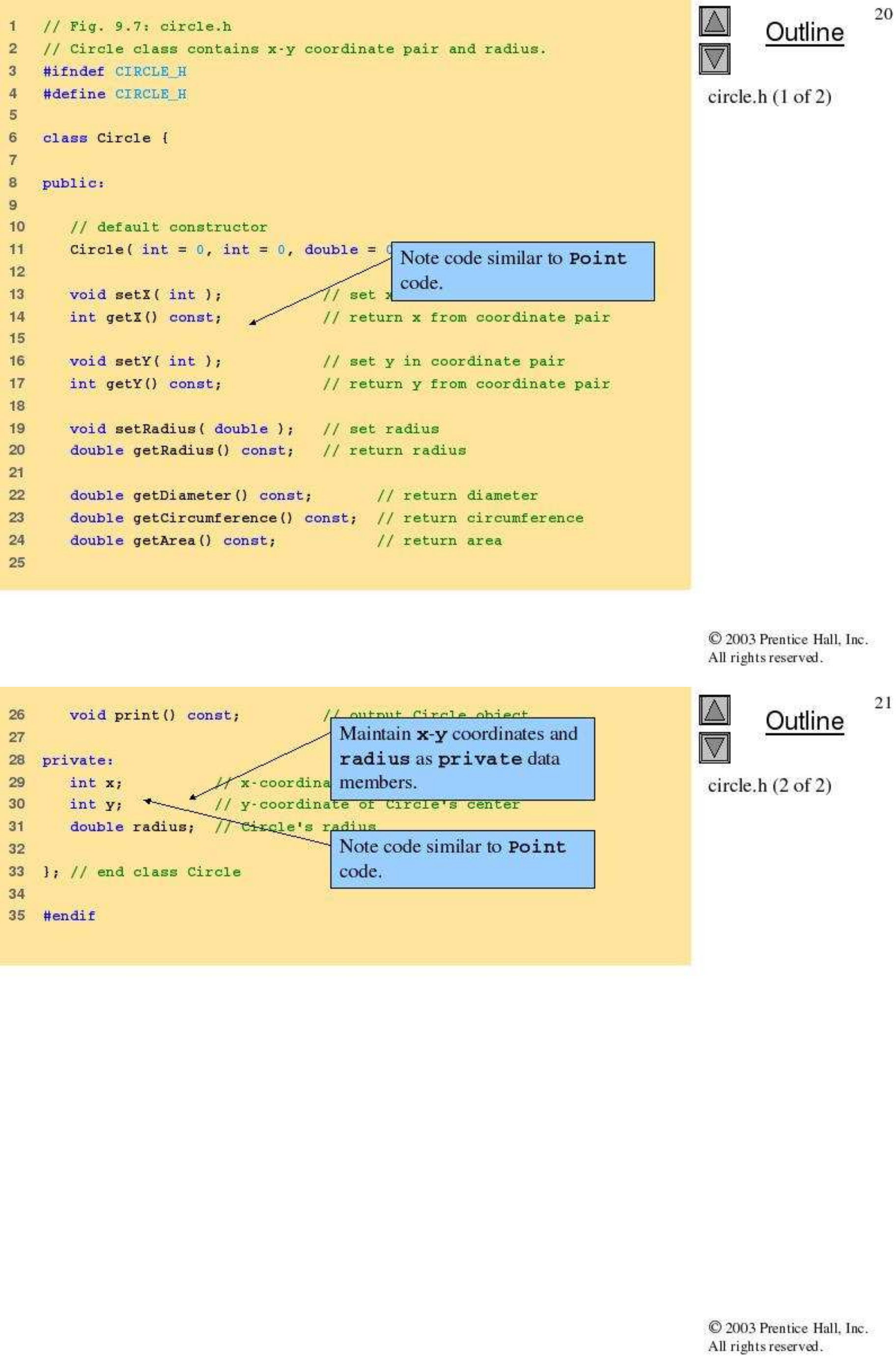
pointtest.cpp
(2 of 2)

pointtest.cpp
output (1 of 1)

© 2003 Prentice Hall, Inc.
All rights reserved.

Figure 4.6: **Point** class test program.

4.4.1 Creating a Circle class without using inheritance



20

Outline

circle.h (1 of 2)

```
// Fig. 9.7: circle.h
// Circle class contains x-y coordinate pair and radius.
#ifndef CIRCLE_H
#define CIRCLE_H

class Circle {

public:

   // default constructor
   Circle( int = 0, int = 0, double = 0

   void setX( int );              // set x
   int getX() const;              // return x from coordinate pair

   void setY( int );              // set y in coordinate pair
   int getY() const;              // return y from coordinate pair

   void setRadius( double );      // set radius
   double getRadius() const;      // return radius

   double getDiameter() const;       // return diameter
   double getCircumference() const;  // return circumference
   double getArea() const;           // return area

```

Note code similar to **Point** code.

21

Outline

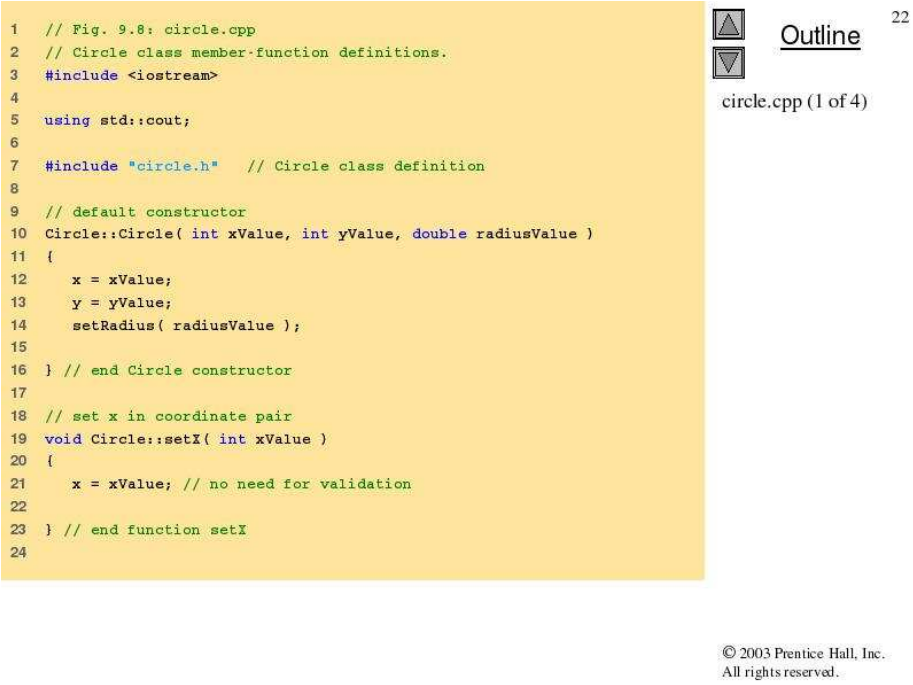
circle.h (2 of 2)

```
   void print() const;            // output Circle object

private:
   int x;                 // x-coordina
   int y;                 // y-coordinate of Circle's center
   double radius;  // Circle's radius

}; // end class Circle

#endif
```

Maintain **x-y** coordinates and **radius** as **private** data members.

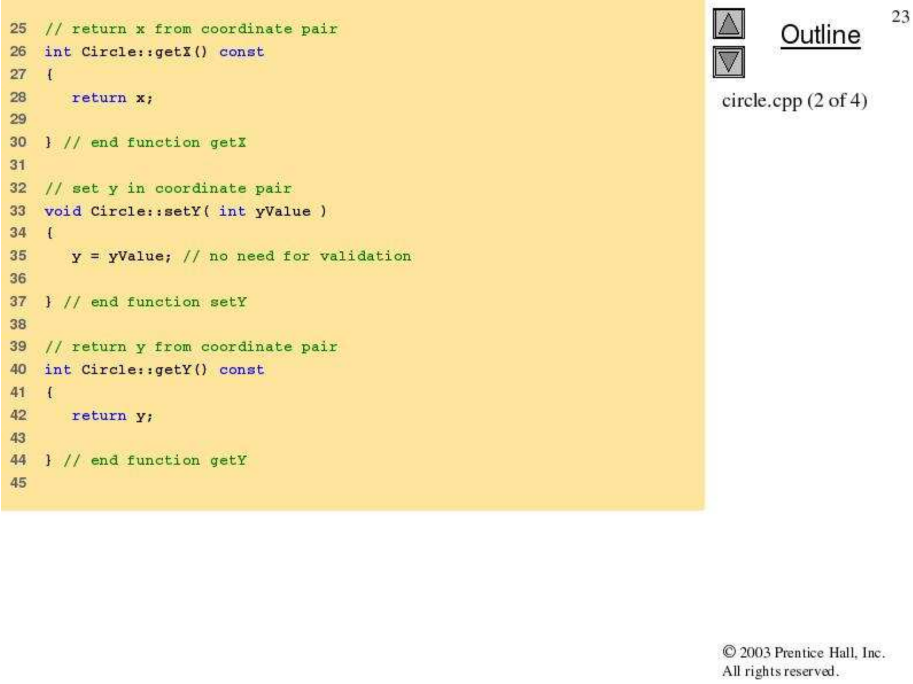
Note code similar to **Point** code.

Figure 4.7: **Circle** class header file.



```
// Fig. 9.8: circle.cpp
// Circle class member-function definitions.
#include <iostream>

using std::cout;

#include "circle.h"   // Circle class definition

// default constructor
Circle::Circle( int xValue, int yValue, double radiusValue )
{
   x = xValue;
   y = yValue;
   setRadius( radiusValue );

} // end Circle constructor

// set x in coordinate pair
void Circle::setX( int xValue )
{
   x = xValue; // no need for validation

} // end function setX

```

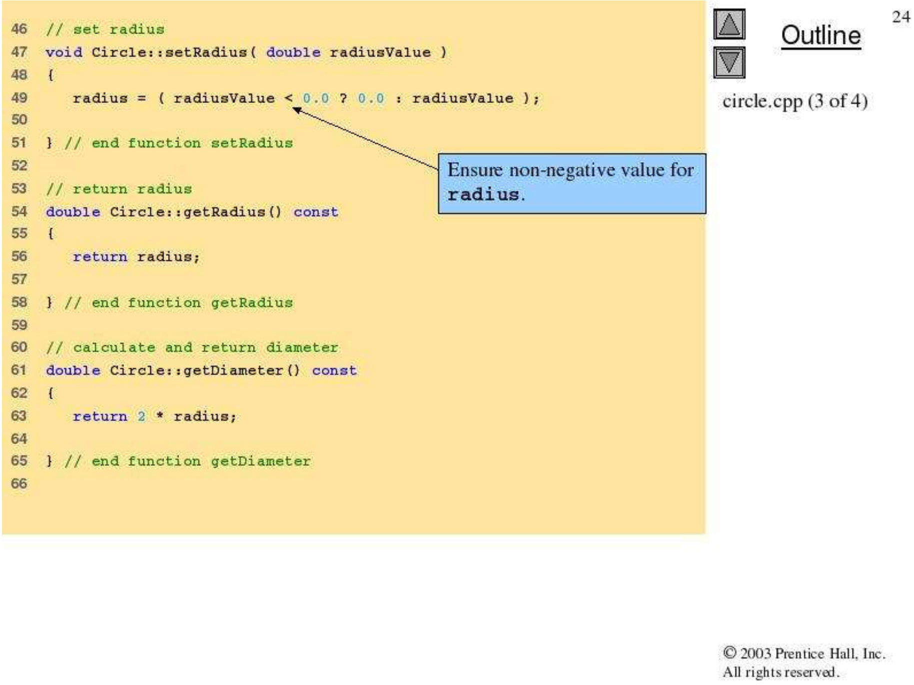
circle.cpp (1 of 4)



```
// return x from coordinate pair
int Circle::getX() const
{
   return x;

} // end function getX

// set y in coordinate pair
void Circle::setY( int yValue )
{
   y = yValue; // no need for validation

} // end function setY

// return y from coordinate pair
int Circle::getY() const
{
   return y;

} // end function getY

```

circle.cpp (2 of 4)

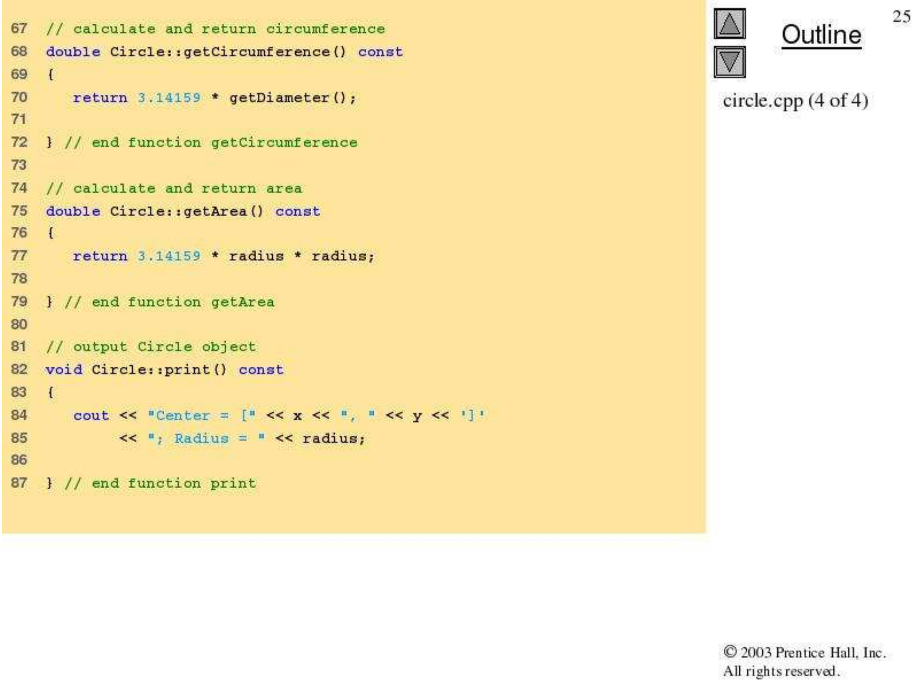
Figure 4.8: **Circle** class contains an xy-coordinate pair and a radius. (part 1 of 2)



```
// set radius
void Circle::setRadius( double radiusValue )
{
   radius = ( radiusValue < 0.0 ? 0.0 : radiusValue );

} // end function setRadius

// return radius
double Circle::getRadius() const
{
   return radius;

} // end function getRadius

// calculate and return diameter
double Circle::getDiameter() const
{
   return 2 * radius;

} // end function getDiameter

```

Ensure non-negative value for **radius**.

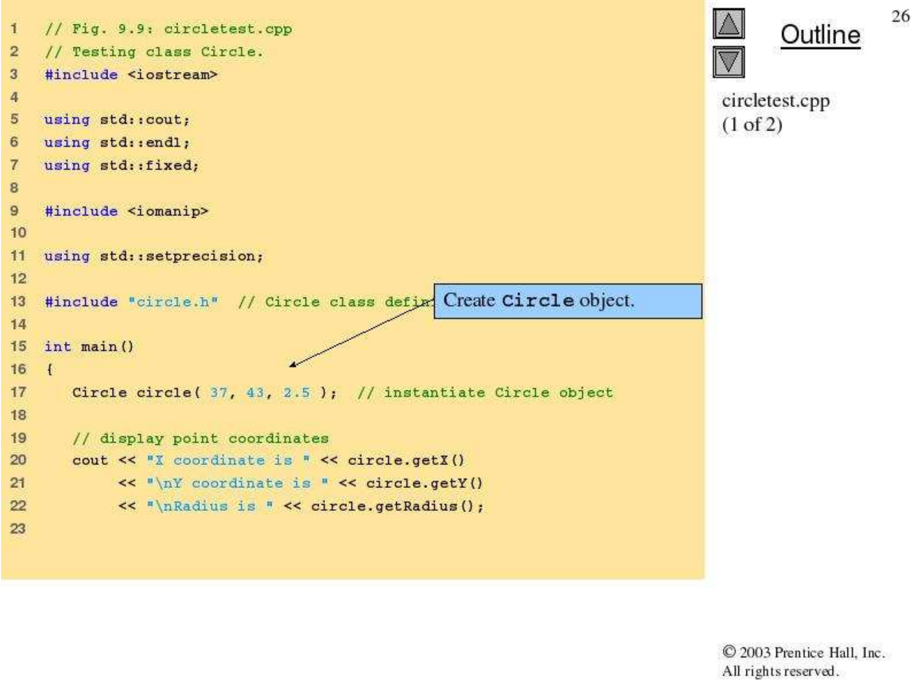
circle.cpp (3 of 4)



```
// calculate and return circumference
double Circle::getCircumference() const
{
   return 3.14159 * getDiameter();

} // end function getCircumference

// calculate and return area
double Circle::getArea() const
{
   return 3.14159 * radius * radius;

} // end function getArea

// output Circle object
void Circle::print() const
{
   cout << "Center = [" << x << ", " << y << ']'
        << "; Radius = " << radius;

} // end function print
```

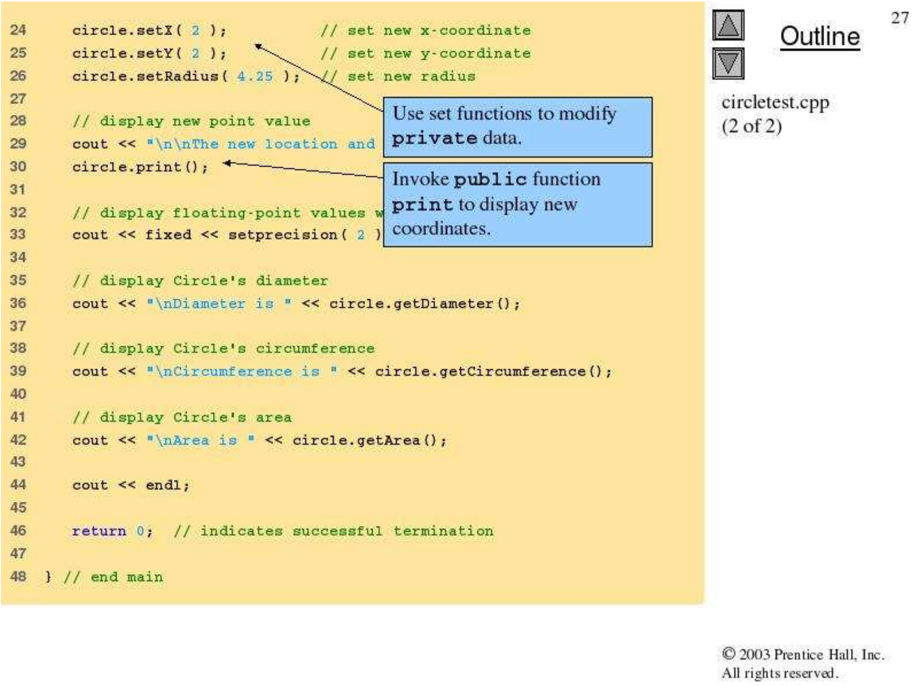
circle.cpp (4 of 4)

Figure 4.9: **Circle** class contains an xy-coordinate pair and a radius. (part 2 of 2)



```
// Fig. 9.9: circletest.cpp
// Testing class Circle.
#include <iostream>

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

#include <iomanip>

using std::setprecision;

#include "circle.h"  // Circle class defi

int main()
{
   Circle circle( 37, 43, 2.5 );  // instantiate Circle object

   // display point coordinates
   cout << "X coordinate is " << circle.getX()
        << "\nY coordinate is " << circle.getY()
        << "\nRadius is " << circle.getRadius();

```

Create **Circle** object.



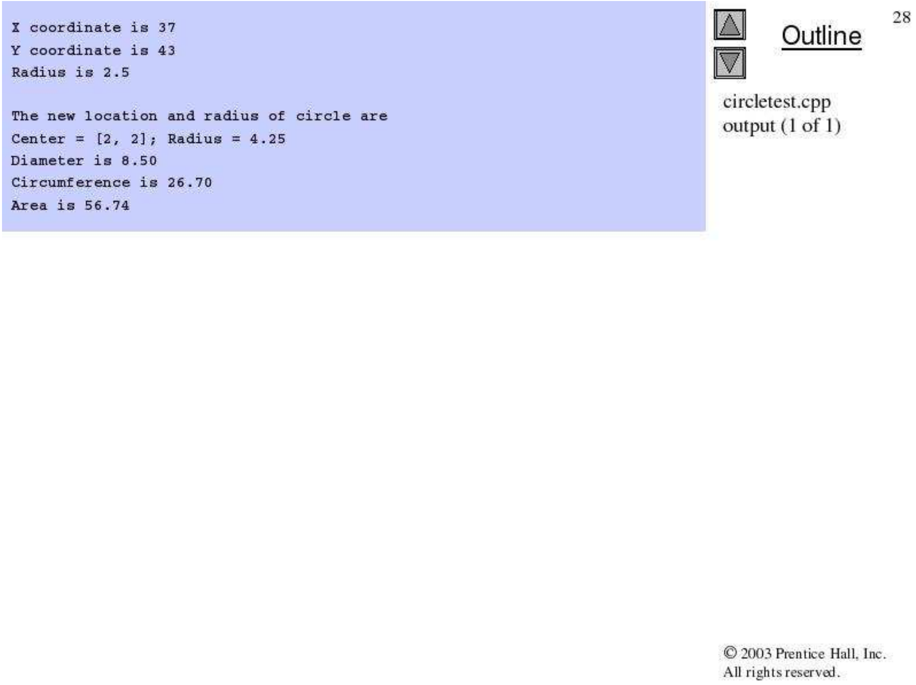
```
   circle.setX( 2 );          // set new x-coordinate
   circle.setY( 2 );          // set new y-coordinate
   circle.setRadius( 4.25 );  // set new radius

   // display new point value
   cout << "\n\nThe new location and
   circle.print();

   // display floating-point values w
   cout << fixed << setprecision( 2 )

   // display Circle's diameter
   cout << "\nDiameter is " << circle.getDiameter();

   // display Circle's circumference
   cout << "\nCircumference is " << circle.getCircumference();

   // display Circle's area
   cout << "\nArea is " << circle.getArea();

   cout << endl;

   return 0;  // indicates successful termination

} // end main
```

Use set functions to modify **private** data.

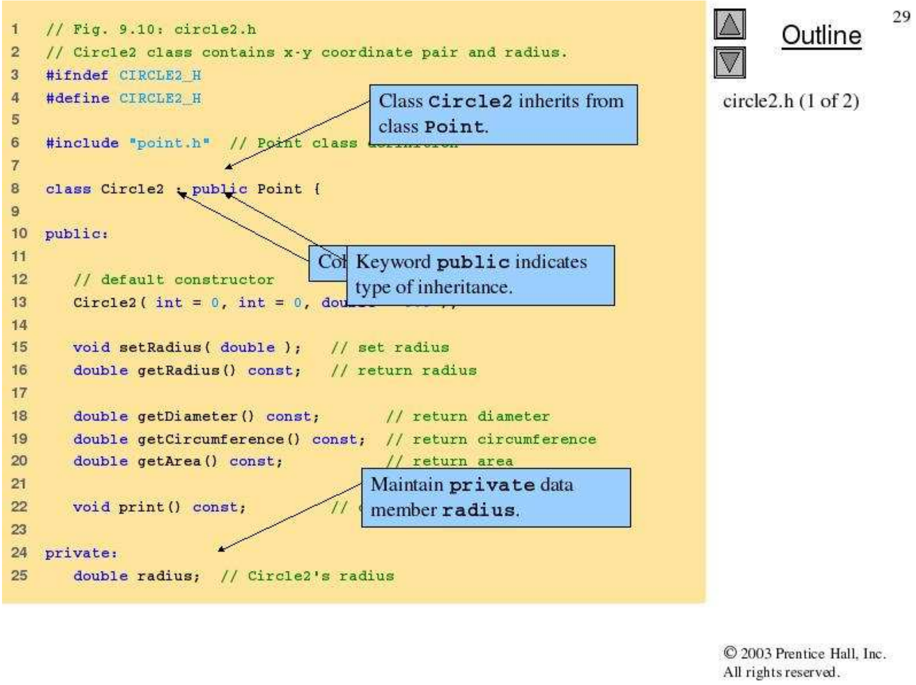
Invoke **public** function **print** to display new coordinates.

Figure 4.10: **Circle** class test program. (part 1 of 2)

4.4.2 Point/Circle Hierarchy using Inheritance



```
X coordinate is 37
Y coordinate is 43
Radius is 2.5

The new location and radius of circle are
Center = [2, 2]; Radius = 4.25
Diameter is 8.50
Circumference is 26.70
Area is 56.74
```



```
// Fig. 9.10: circle2.h
// Circle2 class contains x-y coordinate pair and radius.
#ifndef CIRCLE2_H
#define CIRCLE2_H

#include "point.h"  // Point class definition

class Circle2 : public Point {

public:

   // default constructor
   Circle2( int = 0, int = 0, dou

   void setRadius( double );   // set radius
   double getRadius() const;   // return radius

   double getDiameter() const;       // return diameter
   double getCircumference() const;  // return circumference
   double getArea() const;           // return area

   void print() const;           //

private:
   double radius;  // Circle2's radius
```

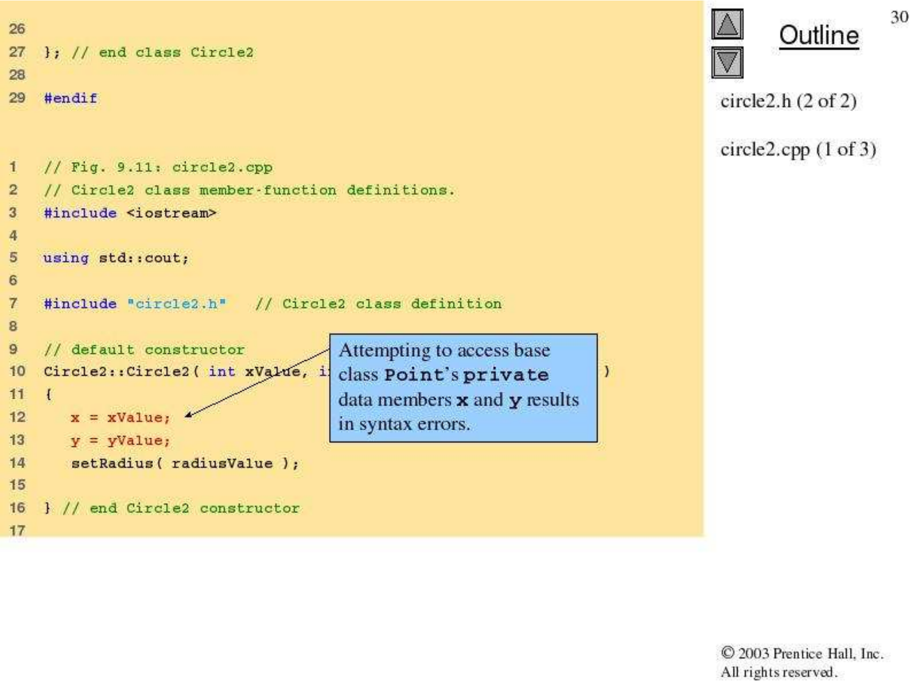
Class **Circle2** inherits from class **Point**.

Co

Keyword **public** indicates type of inheritance.

Maintain **private** data member **radius**.

Figure 4.11: **Circle** class test program. (part 2 of 2) and **Circle2** class header file. (part 1 of 2)



30

Outline

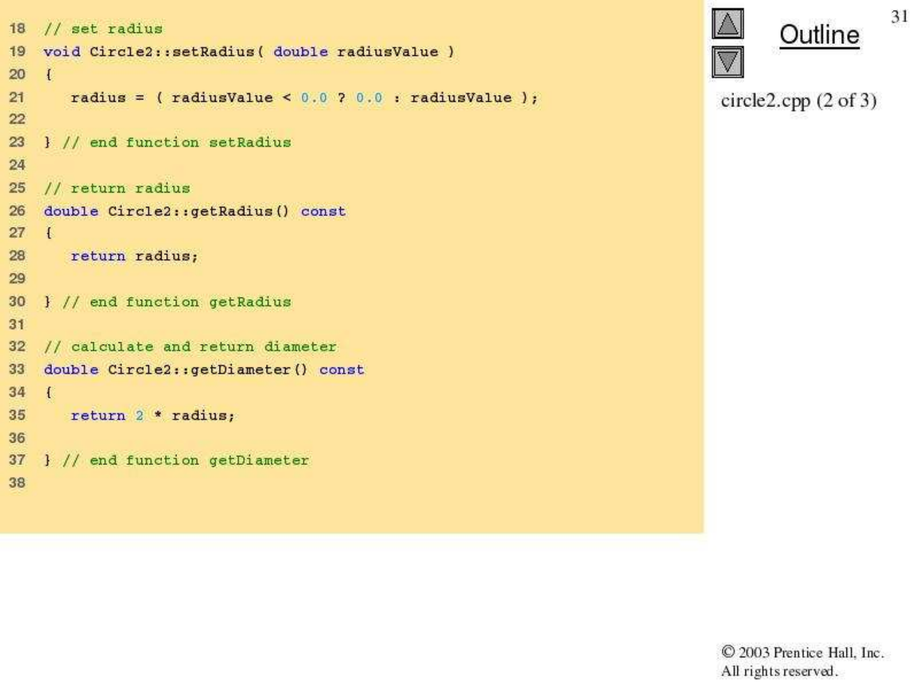
circle2.h (2 of 2)

circle2.cpp (1 of 3)

```

}; // end class Circle2

#endif
```

```
// Fig. 9.11: circle2.cpp
// Circle2 class member-function definitions.
#include <iostream>

using std::cout;

#include "circle2.h"   // Circle2 class definition

// default constructor
Circle2::Circle2( int xValue, i                              )
{
   x = xValue;
   y = yValue;
   setRadius( radiusValue );

} // end Circle2 constructor

```

Attempting to access base class **Point**'s **private** data members **x** and **y** results in syntax errors.



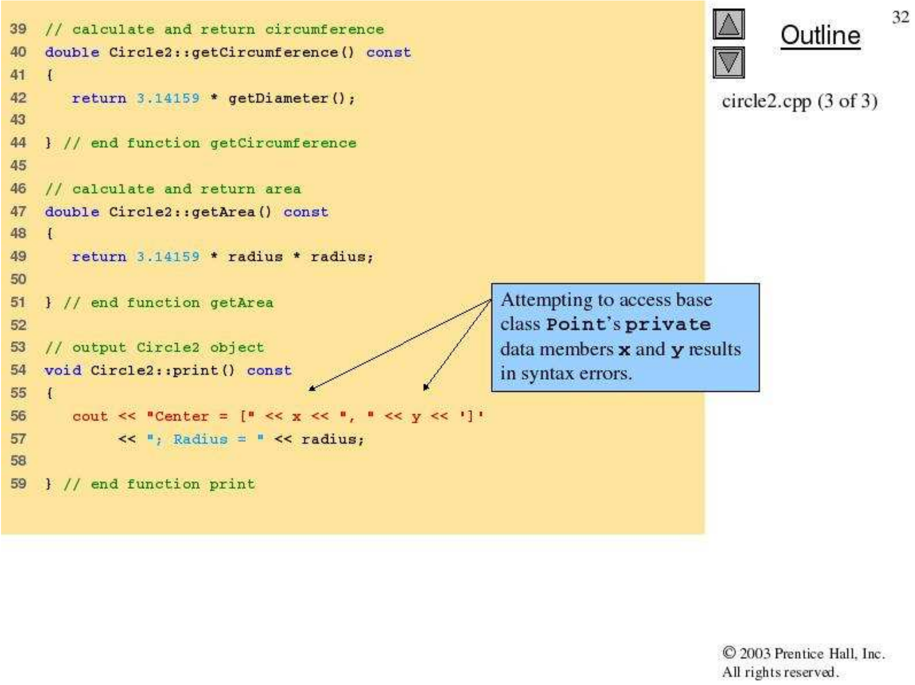
31

Outline

circle2.cpp (2 of 3)

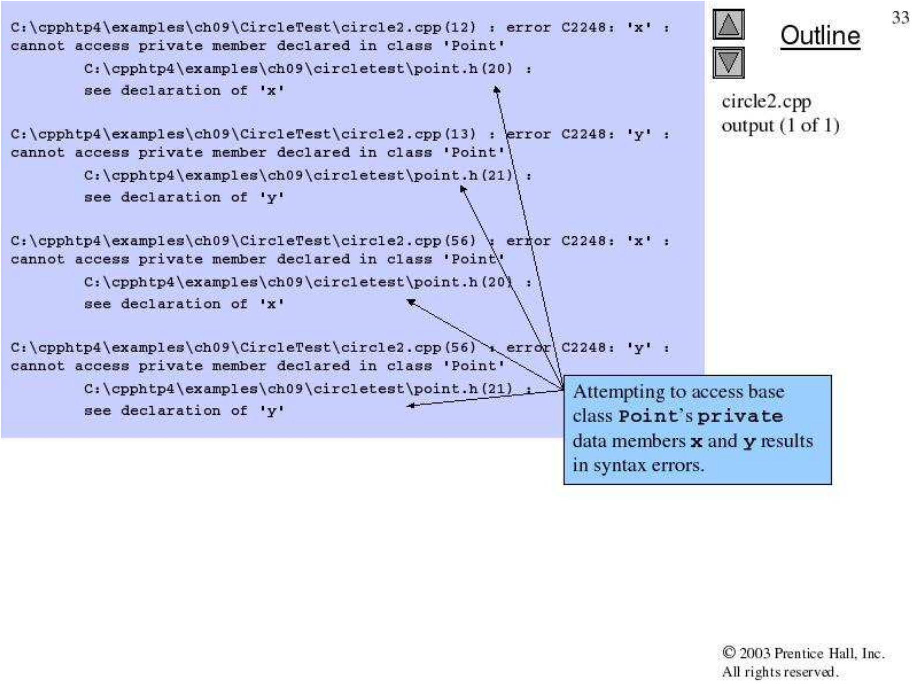
```
// set radius
void Circle2::setRadius( double radiusValue )
{
   radius = ( radiusValue < 0.0 ? 0.0 : radiusValue );

} // end function setRadius

// return radius
double Circle2::getRadius() const
{
   return radius;

} // end function getRadius

// calculate and return diameter
double Circle2::getDiameter() const
{
   return 2 * radius;

} // end function getDiameter

```

Figure 4.12: **Circle2** class header file (part 2 of 2) and Private base-class data can not be accessed from derived class. (part 1 of 2)



```
39  // calculate and return circumference
40  double Circle2::getCircumference() const
41  {
42     return 3.14159 * getDiameter();
43
44  } // end function getCircumference
45
46  // calculate and return area
47  double Circle2::getArea() const
48  {
49     return 3.14159 * radius * radius;
50
51  } // end function getArea
52
53  // output Circle2 object
54  void Circle2::print() const
55  {
56     cout << "Center = [" << x << ", " << y << ']'
57          << "; Radius = " << radius;
58
59  } // end function print
```

Attempting to access base class `Point`'s `private` data members `x` and `y` results in syntax errors.

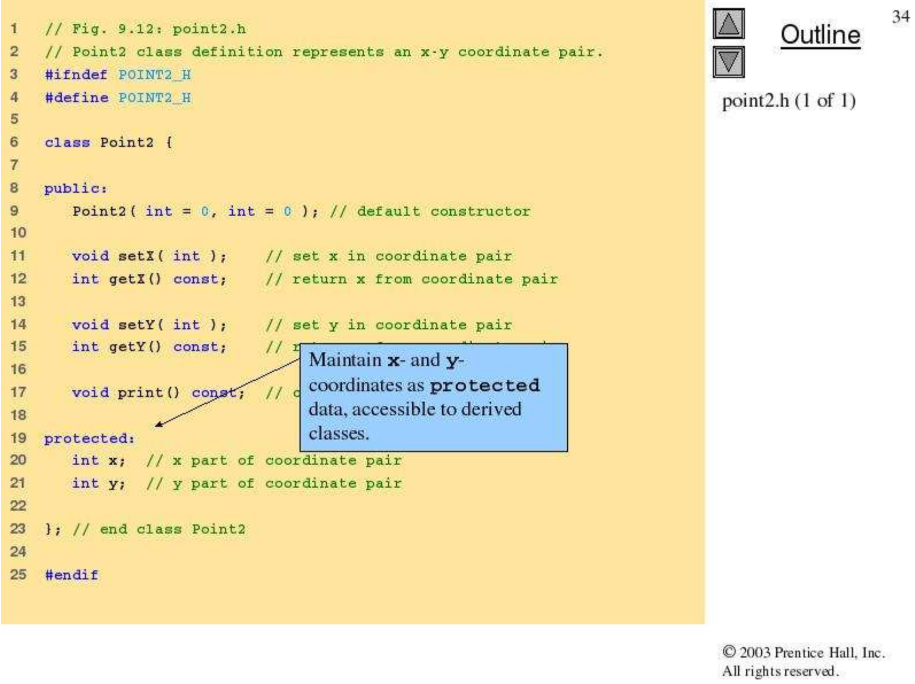


```
C:\cpphtp4\examples\ch09\CircleTest\circle2.cpp(12) : error C2248: 'x' :
cannot access private member declared in class 'Point'
        C:\cpphtp4\examples\ch09\circletest\point.h(20) :
        see declaration of 'x'

C:\cpphtp4\examples\ch09\CircleTest\circle2.cpp(13) : error C2248: 'y' :
cannot access private member declared in class 'Point'
        C:\cpphtp4\examples\ch09\circletest\point.h(21) :
        see declaration of 'y'

C:\cpphtp4\examples\ch09\CircleTest\circle2.cpp(56) : error C2248: 'x' :
cannot access private member declared in class 'Point'
        C:\cpphtp4\examples\ch09\circletest\point.h(20) :
        see declaration of 'x'

C:\cpphtp4\examples\ch09\CircleTest\circle2.cpp(56) : error C2248: 'y' :
cannot access private member declared in class 'Point'
        C:\cpphtp4\examples\ch09\circletest\point.h(21) :
        see declaration of 'y'
```

Attempting to access base class `Point`'s `private` data members `x` and `y` results in syntax errors.

Figure 4.13: Private base-class data can not be accessed from derived class. (part 2 of 2)

4.4.3 Point/Circle Hierarchy using protected data



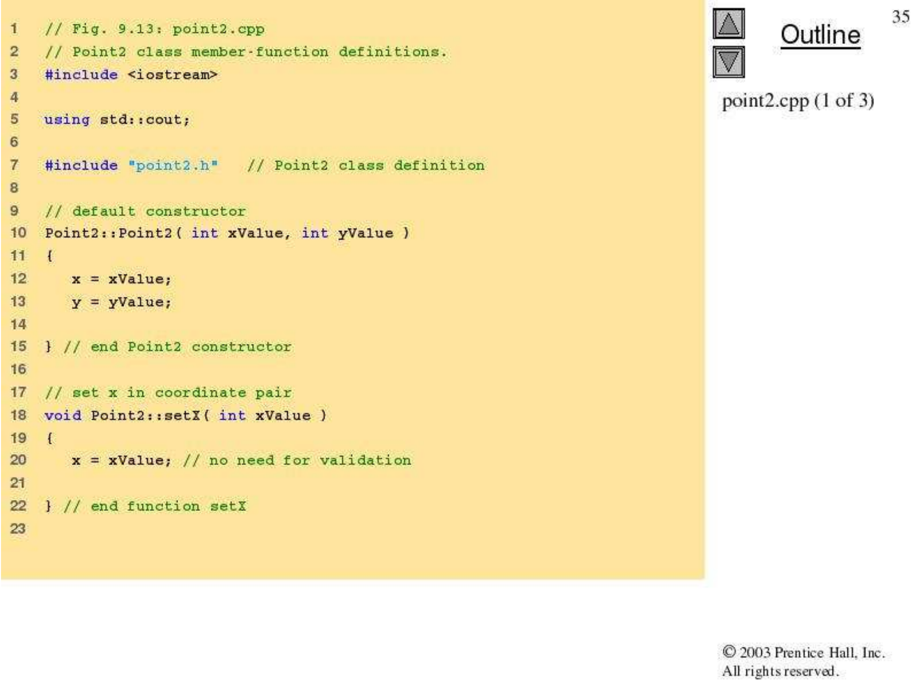
```
// Fig. 9.12: point2.h
// Point2 class definition represents an x-y coordinate pair.
#ifndef POINT2_H
#define POINT2_H

class Point2 {

public:
   Point2( int = 0, int = 0 ); // default constructor

   void setX( int );     // set x in coordinate pair
   int getX() const;     // return x from coordinate pair

   void setY( int );     // set y in coordinate pair
   int getY() const;     // r

   void print() const;   // c

protected:
   int x;  // x part of coordinate pair
   int y;  // y part of coordinate pair

}; // end class Point2

#endif
```

Maintain **x**- and **y**-coordinates as **protected** data, accessible to derived classes.

34

Outline

point2.h (1 of 1)



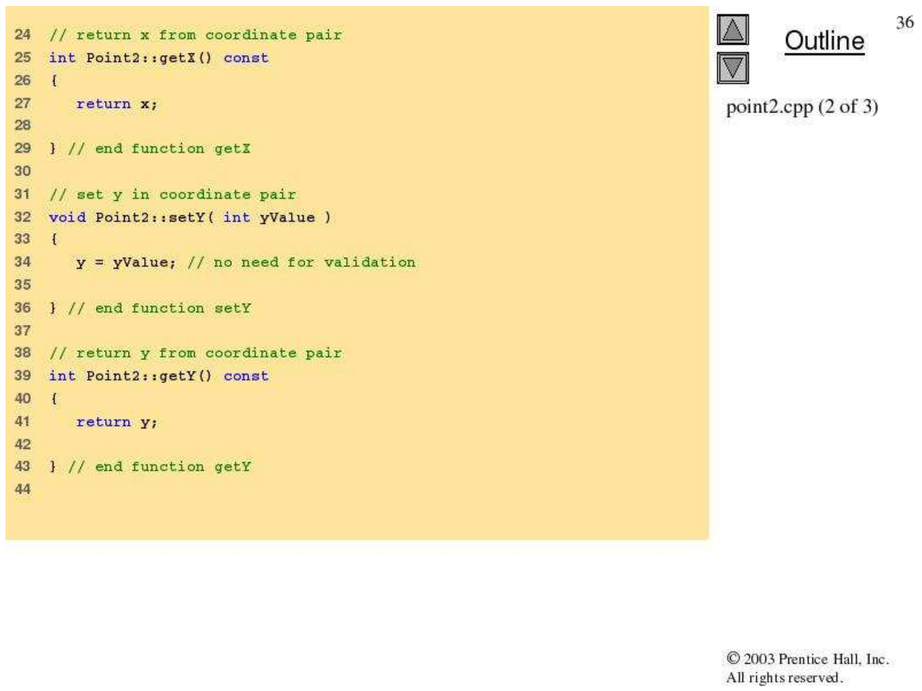
```
// Fig. 9.13: point2.cpp
// Point2 class member-function definitions.
#include <iostream>

using std::cout;

#include "point2.h"   // Point2 class definition

// default constructor
Point2::Point2( int xValue, int yValue )
{
   x = xValue;
   y = yValue;

} // end Point2 constructor

// set x in coordinate pair
void Point2::setX( int xValue )
{
   x = xValue; // no need for validation

} // end function setX

```

35

Outline

point2.cpp (1 of 3)

Figure 4.14: **Point2** class header file.

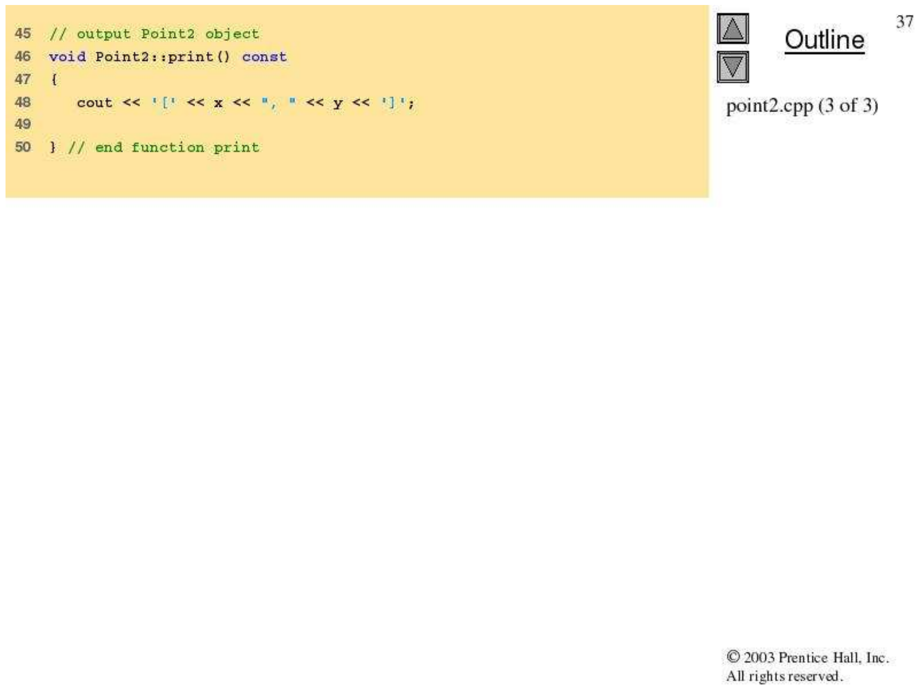


36

Outline

point2.cpp (2 of 3)

```
// return x from coordinate pair
int Point2::getX() const
{
   return x;

} // end function getX

// set y in coordinate pair
void Point2::setY( int yValue )
{
   y = yValue; // no need for validation

} // end function setY

// return y from coordinate pair
int Point2::getY() const
{
   return y;

} // end function getY

```



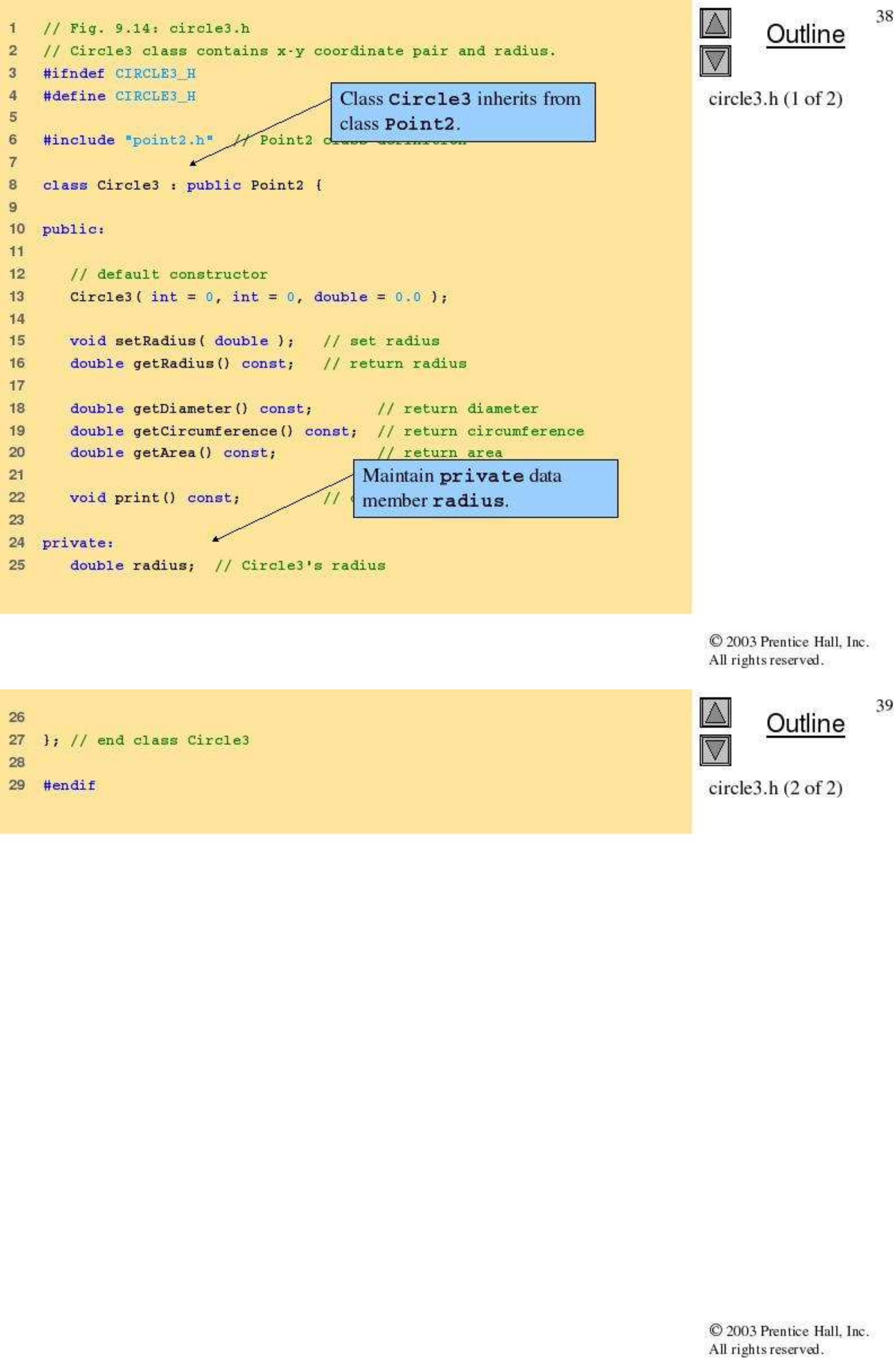
37

Outline

point2.cpp (3 of 3)

```
// output Point2 object
void Point2::print() const
{
   cout << '[' << x << ", " << y << ']';

} // end function print
```

Figure 4.15: **Point2** class represents an xy-coordinate pair as **protected** data.



```
// Fig. 9.14: circle3.h
// Circle3 class contains x-y coordinate pair and radius.
#ifndef CIRCLE3_H
#define CIRCLE3_H

#include "point2.h"  // Point2 class definition

class Circle3 : public Point2 {

public:

   // default constructor
   Circle3( int = 0, int = 0, double = 0.0 );

   void setRadius( double );   // set radius
   double getRadius() const;   // return radius

   double getDiameter() const;       // return diameter
   double getCircumference() const;  // return circumference
   double getArea() const;           // return area

   void print() const;               //

private:
   double radius;  // Circle3's radius
```

Class **Circle3** inherits from class **Point2**.

Maintain **private** data member **radius**.

38

Outline

circle3.h (1 of 2)

© 2003 Prentice Hall, Inc.
All rights reserved.

```

}; // end class Circle3

#endif
```

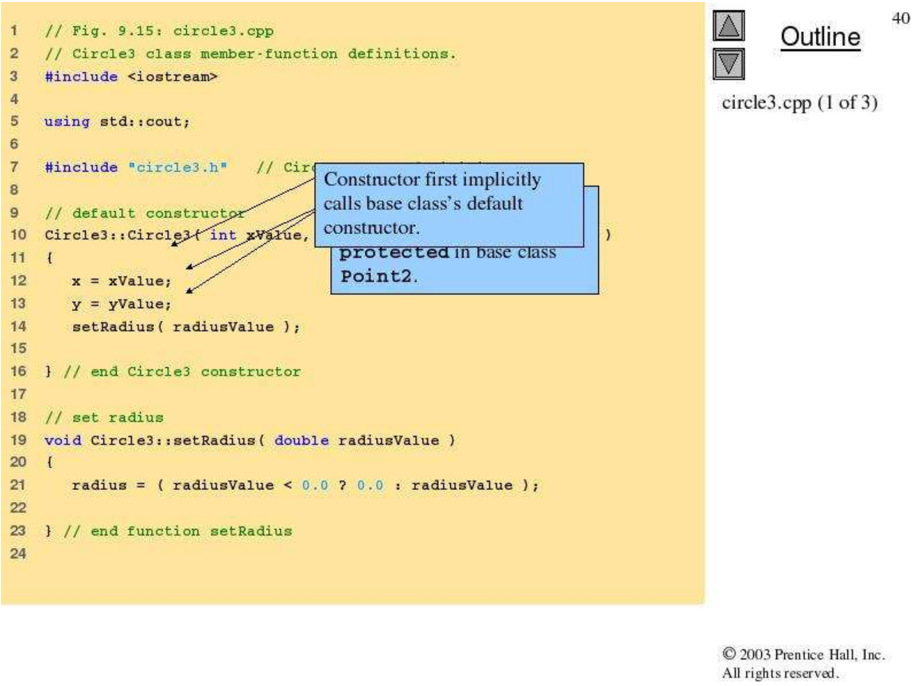
39

Outline

circle3.h (2 of 2)

© 2003 Prentice Hall, Inc.
All rights reserved.

Figure 4.16: **Circle3** class header file.



40

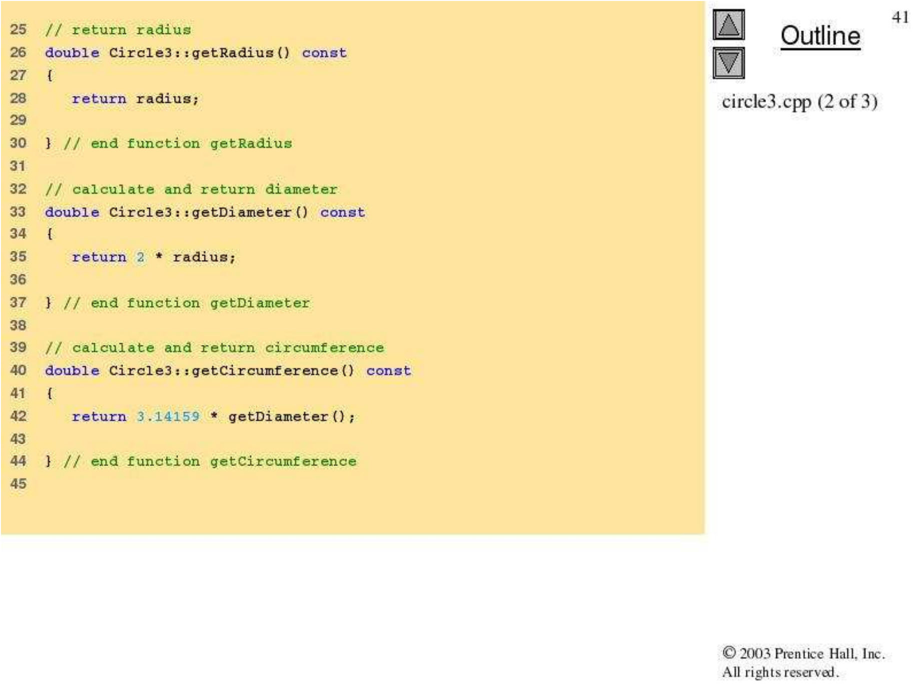
Outline

circle3.cpp (1 of 3)

```
// Fig. 9.15: circle3.cpp
// Circle3 class member-function definitions.
#include <iostream>

using std::cout;

#include "circle3.h"   // Cir

// default constructor
Circle3::Circle3( int xValue,                                )
{
   x = xValue;
   y = yValue;
   setRadius( radiusValue );

} // end Circle3 constructor

// set radius
void Circle3::setRadius( double radiusValue )
{
   radius = ( radiusValue < 0.0 ? 0.0 : radiusValue );

} // end function setRadius

```

Constructor first implicitly calls base class's default constructor.

protected in base class **Point2**.



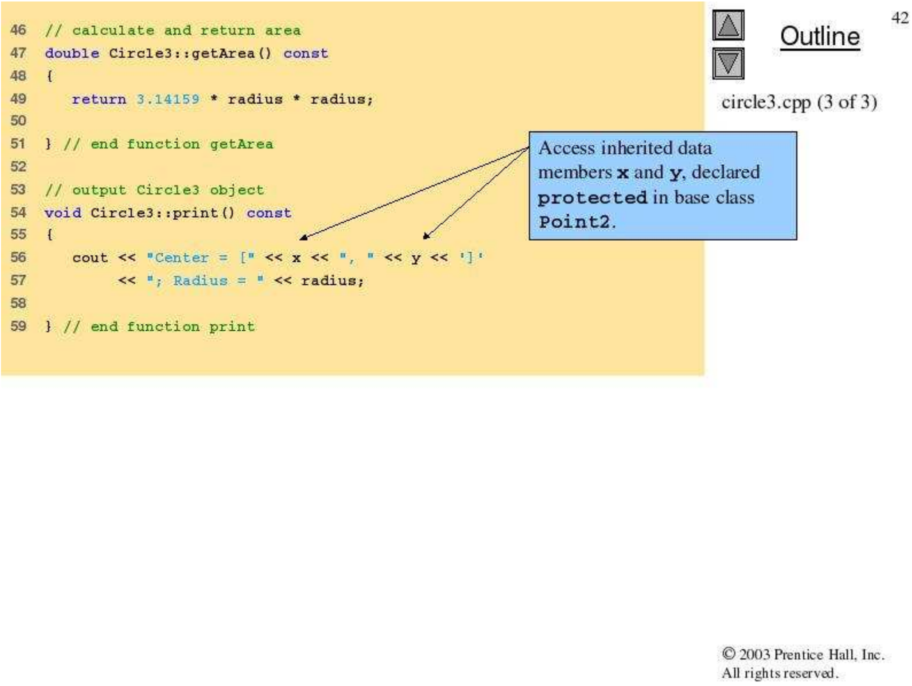
41

Outline

circle3.cpp (2 of 3)

```
// return radius
double Circle3::getRadius() const
{
   return radius;

} // end function getRadius

// calculate and return diameter
double Circle3::getDiameter() const
{
   return 2 * radius;

} // end function getDiameter

// calculate and return circumference
double Circle3::getCircumference() const
{
   return 3.14159 * getDiameter();

} // end function getCircumference

```

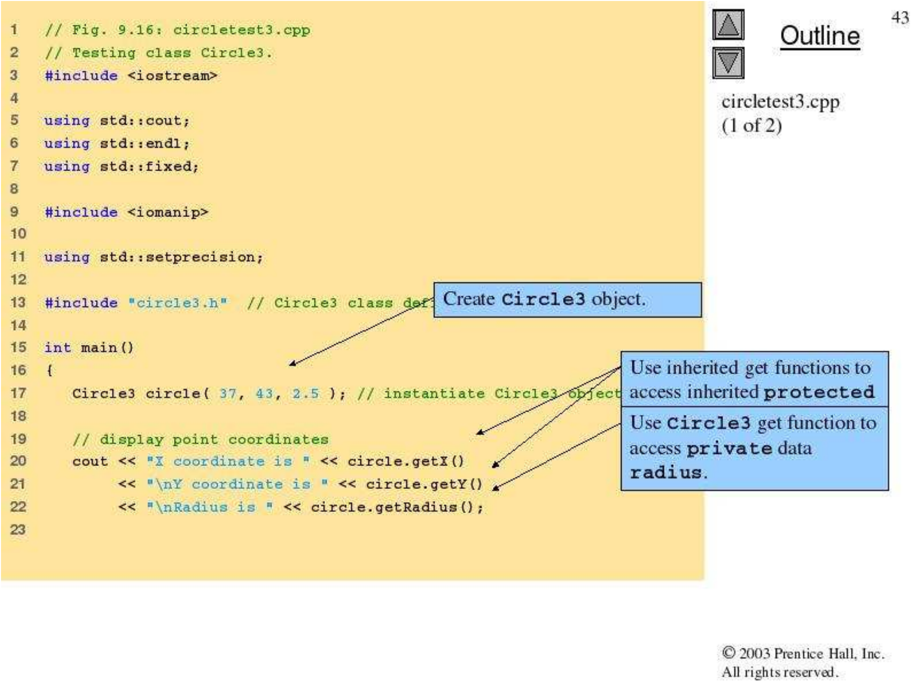
Figure 4.17: **Circle3** class that inherits from class **Point2**.



```
// calculate and return area
double Circle3::getArea() const
{
   return 3.14159 * radius * radius;

} // end function getArea

// output Circle3 object
void Circle3::print() const
{
   cout << "Center = [" << x << ", " << y << ']'
        << "; Radius = " << radius;

} // end function print
```

Access inherited data members **x** and **y**, declared **protected** in base class **Point2**.

circle3.cpp (3 of 3)



```
// Fig. 9.16: circletest3.cpp
// Testing class Circle3.
#include <iostream>

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

#include <iomanip>

using std::setprecision;

#include "circle3.h"  // Circle3 class def

int main()
{
   Circle3 circle( 37, 43, 2.5 ); // instantiate Circle3 object

   // display point coordinates
   cout << "X coordinate is " << circle.getX()
        << "\nY coordinate is " << circle.getY()
        << "\nRadius is " << circle.getRadius();

```

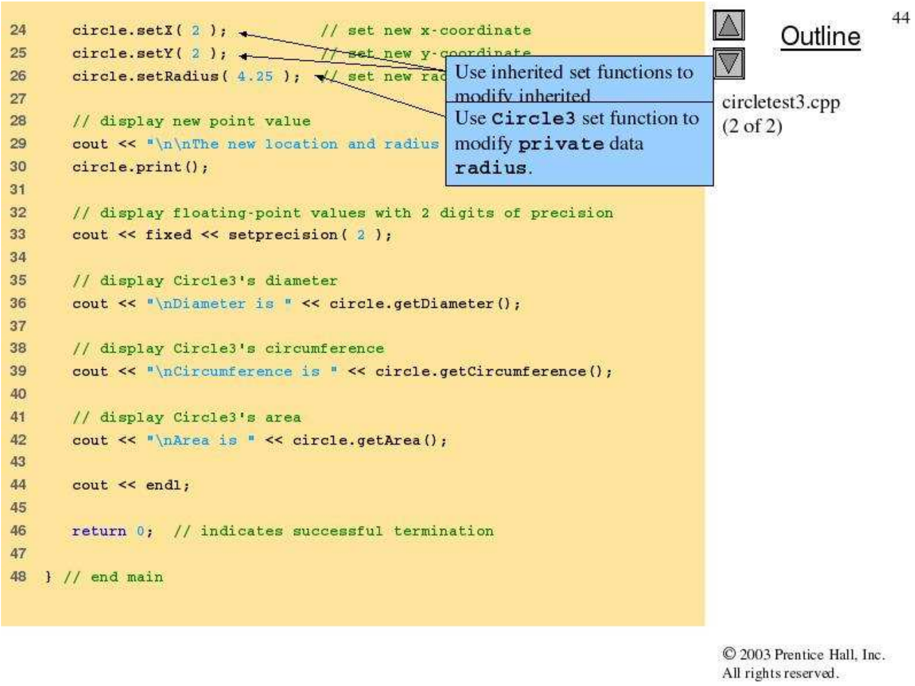
Create **Circle3** object.

Use inherited get functions to access inherited **protected**

Use **Circle3** get function to access **private** data **radius**.

circletest3.cpp (1 of 2)

Figure 4.18: Protected base-class data can be accessed from derived class. (part 1 of 2)



44

Outline

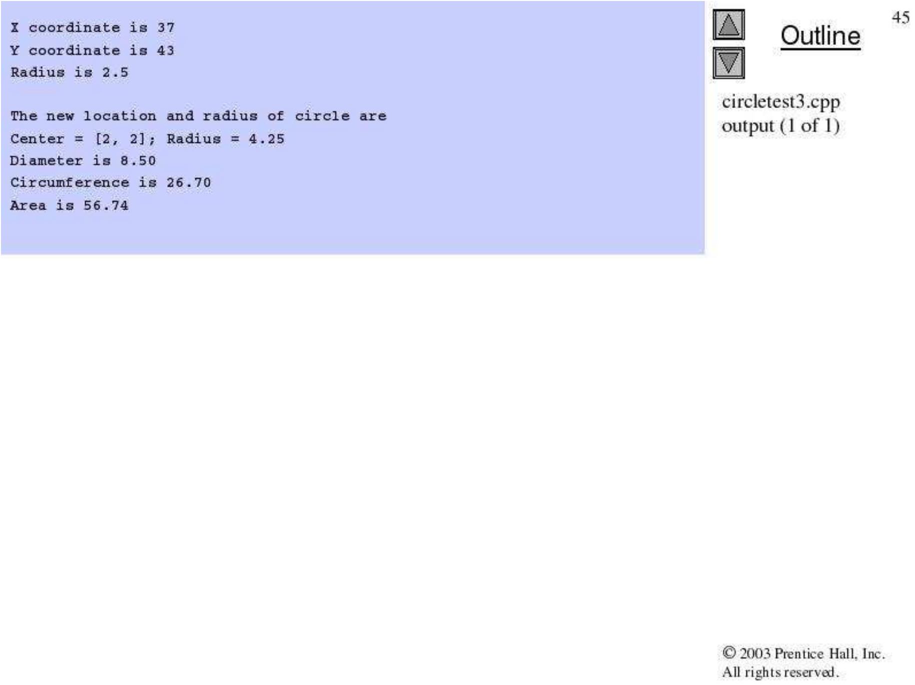
```
   circle.setX( 2 );            // set new x-coordinate
   circle.setY( 2 );            // set new y-coordinate
   circle.setRadius( 4.25 );   // set new rad

   // display new point value
   cout << "\n\nThe new location and radius
   circle.print();

   // display floating-point values with 2 digits of precision
   cout << fixed << setprecision( 2 );

   // display Circle3's diameter
   cout << "\nDiameter is " << circle.getDiameter();

   // display Circle3's circumference
   cout << "\nCircumference is " << circle.getCircumference();

   // display Circle3's area
   cout << "\nArea is " << circle.getArea();

   cout << endl;

   return 0;  // indicates successful termination

} // end main
```

Use inherited set functions to modify inherited

Use **Circle3** set function to modify **private** data **radius**.

circletest3.cpp
(2 of 2)

© 2003 Prentice Hall, Inc.
All rights reserved.



45

Outline

```
X coordinate is 37
Y coordinate is 43
Radius is 2.5

The new location and radius of circle are
Center = [2, 2]; Radius = 4.25
Diameter is 8.50
Circumference is 26.70
Area is 56.74
```

circletest3.cpp
output (1 of 1)

© 2003 Prentice Hall, Inc.
All rights reserved.

Figure 4.19: Protected base-class data can be accessed from derived class. (part 2 of 2)

  - Derived classes can modify values directly
  - Slight increase in performance; avoid set/get function call overhead

- Disadvantages
  - No validity checking; derived class can assign illegal value
  - Implementation dependent
    - Derived class member functions more likely dependent on base class implementation
    - Base class implementation changes may result in derived class modifications; fragile (brittle) software

4.4.4 Point/Circle Hierarchy using private data

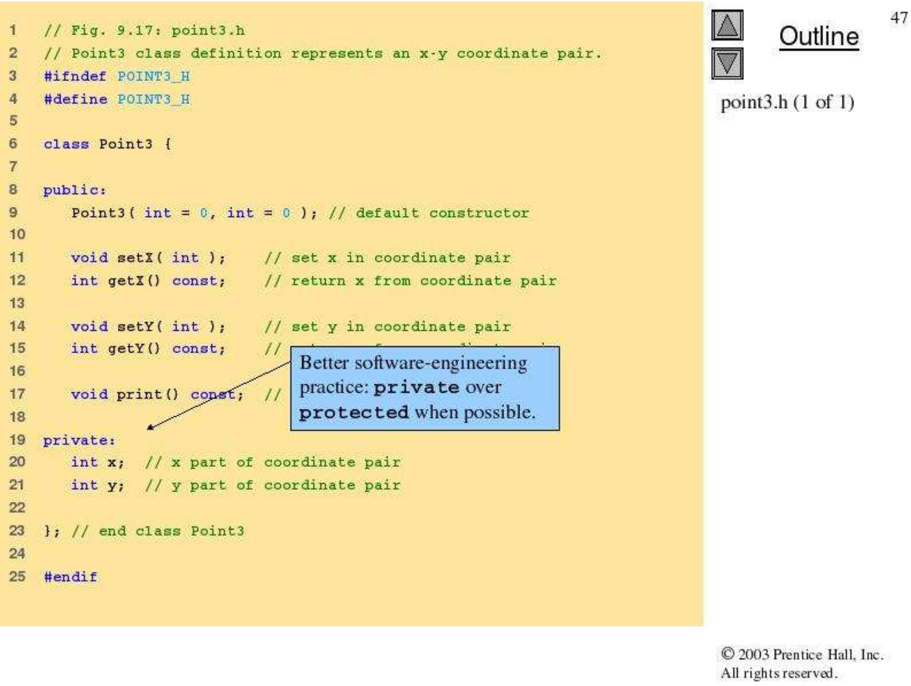
Class **Point3** (Figs. 4.20-4.21) declares data members **x** and **y** as **private** and exposes member functions **setX, getX, setY, getY** and **print** for manipulating these values.

4.5 Case Study: Three-Level Inheritance Hierarchy

Three level point/circle/cylinder hierarchy

- Point
  - x-y coordinate pair
- Circle
  - x-y coordinate pair
  - Radius
- Cylinder
  - x-y coordinate pair
  - Radius
  - Height

Derive class **Cylinder** from class **Circle4**. Class **Cylinder** should redefine member functions **getArea** and **print** member functions. Figs. 4.26-4.27 present class **Cylinder**, which inherits from class **Circle4**. We were able to develop classes Circle4 and Cylinder much more quickly by using inheritance than if we had developed these classes "from scratch". Inheritance avoids duplicating code and the associated code-maintenance problems.



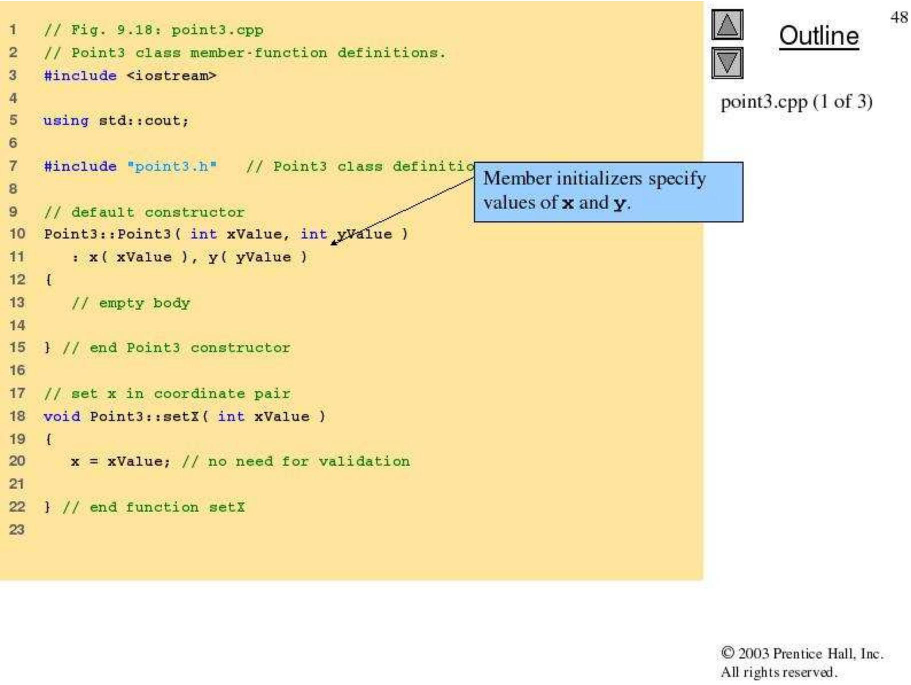
47

Outline

point3.h (1 of 1)

```
// Fig. 9.17: point3.h
// Point3 class definition represents an x-y coordinate pair.
#ifndef POINT3_H
#define POINT3_H

class Point3 {

public:
   Point3( int = 0, int = 0 ); // default constructor

   void setX( int );     // set x in coordinate pair
   int getX() const;     // return x from coordinate pair

   void setY( int );     // set y in coordinate pair
   int getY() const;     //

   void print() const;   //

private:
   int x;  // x part of coordinate pair
   int y;  // y part of coordinate pair

}; // end class Point3

#endif
```

Better software-engineering practice: **private** over **protected** when possible.



48

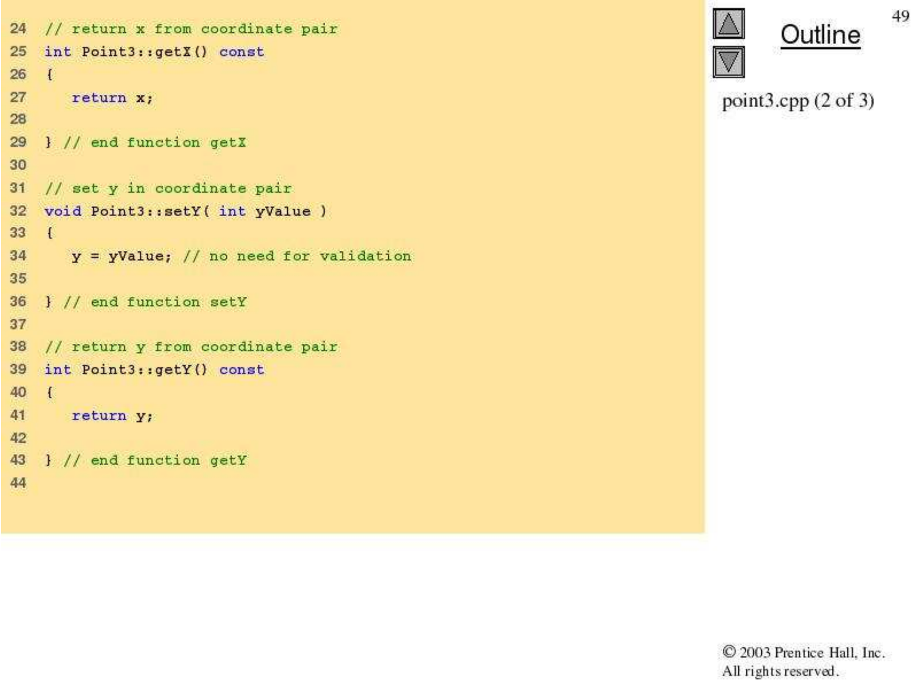
Outline

point3.cpp (1 of 3)

```
// Fig. 9.18: point3.cpp
// Point3 class member-function definitions.
#include <iostream>

using std::cout;

#include "point3.h"   // Point3 class definitio

// default constructor
Point3::Point3( int xValue, int yValue )
   : x( xValue ), y( yValue )
{
   // empty body

} // end Point3 constructor

// set x in coordinate pair
void Point3::setX( int xValue )
{
   x = xValue; // no need for validation

} // end function setX

```

Member initializers specify values of **x** and **y**.

Figure 4.20: **Point3** class header file. Point/Circle Hierarchy Using **private** Data



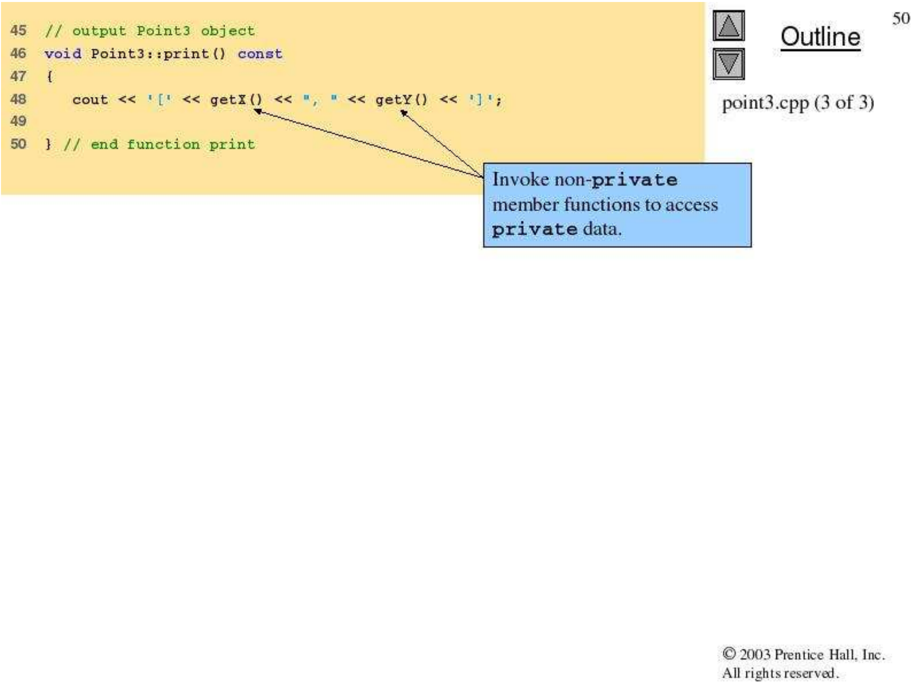
49

Outline

point3.cpp (2 of 3)

```cpp
// return x from coordinate pair
int Point3::getX() const
{
   return x;

} // end function getX

// set y in coordinate pair
void Point3::setY( int yValue )
{
   y = yValue; // no need for validation

} // end function setY

// return y from coordinate pair
int Point3::getY() const
{
   return y;

} // end function getY

```

© 2003 Prentice Hall, Inc.
All rights reserved.



50

Outline

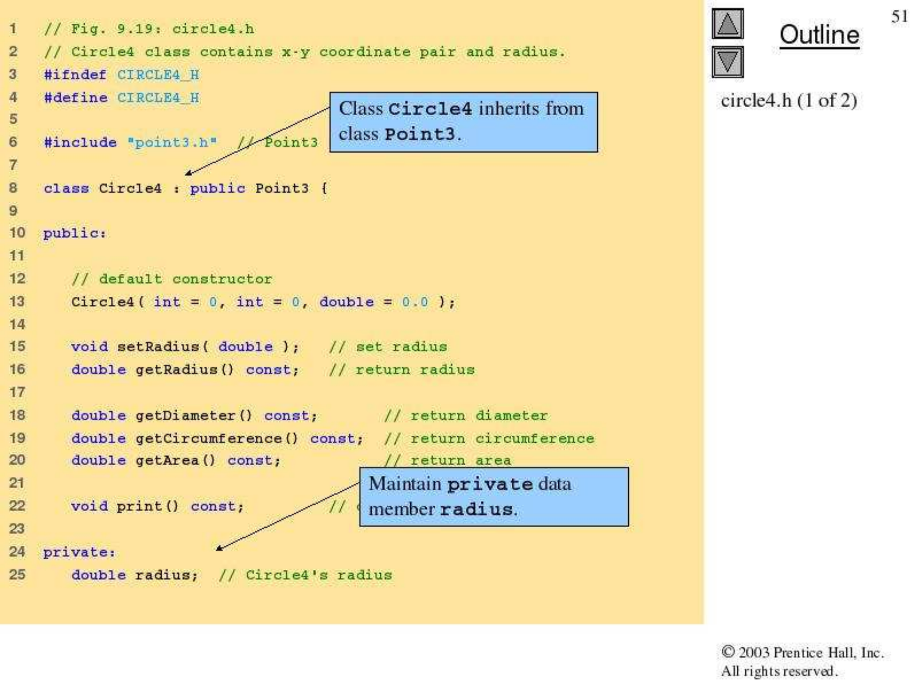
point3.cpp (3 of 3)

```cpp
// output Point3 object
void Point3::print() const
{
   cout << '[' << getX() << ", " << getY() << ']';

} // end function print
```

Invoke non-**private** member functions to access **private** data.

© 2003 Prentice Hall, Inc.
All rights reserved.

Figure 4.21: **Point3** class uses member functions to manipulate its **private** data.



```
// Fig. 9.19: circle4.h
// Circle4 class contains x-y coordinate pair and radius.
#ifndef CIRCLE4_H
#define CIRCLE4_H

#include "point3.h"  // Point3

class Circle4 : public Point3 {

public:

   // default constructor
   Circle4( int = 0, int = 0, double = 0.0 );

   void setRadius( double );   // set radius
   double getRadius() const;   // return radius

   double getDiameter() const;       // return diameter
   double getCircumference() const;  // return circumference
   double getArea() const;           // return area

   void print() const;               // 

private:
   double radius;  // Circle4's radius
```

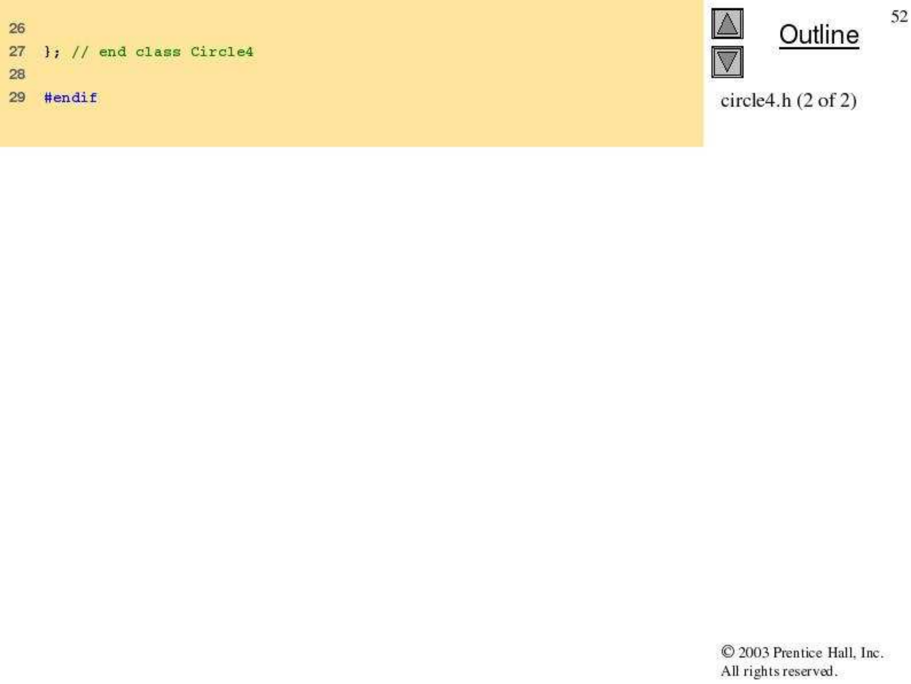
Class **Circle4** inherits from class **Point3**.

Maintain **private** data member **radius**.

51

Outline

circle4.h (1 of 2)



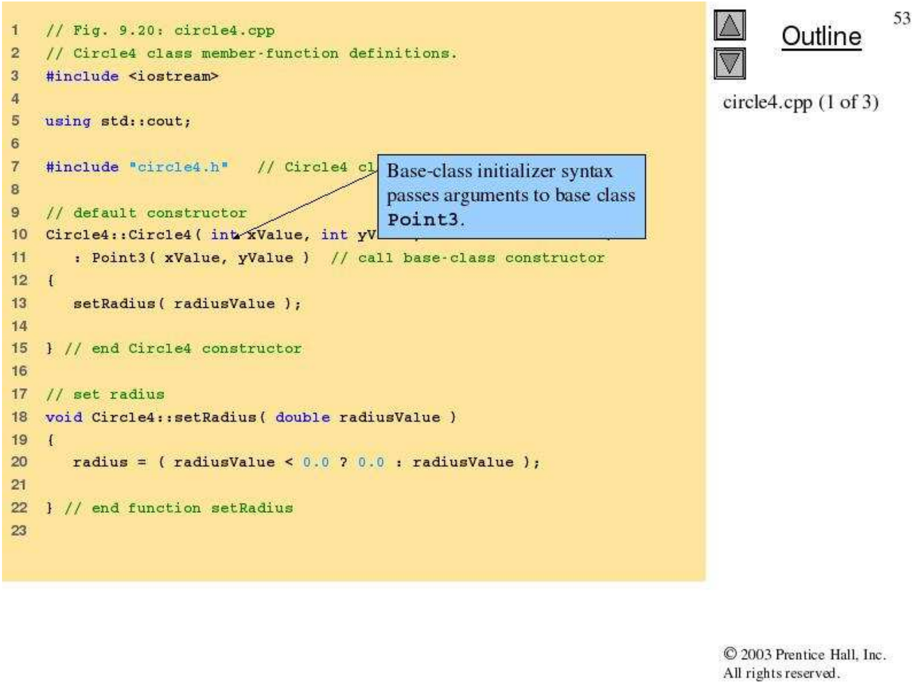
```

}; // end class Circle4

#endif
```

52

Outline

circle4.h (2 of 2)

Figure 4.22: **Circle4** class header file.



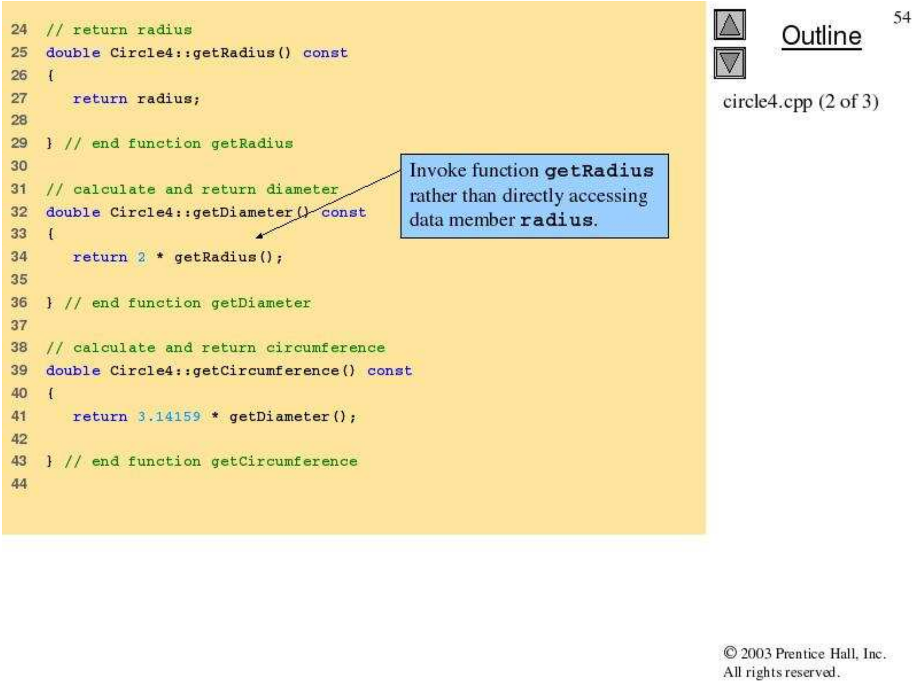
53

Outline

circle4.cpp (1 of 3)

```
// Fig. 9.20: circle4.cpp
// Circle4 class member-function definitions.
#include <iostream>

using std::cout;

#include "circle4.h"   // Circle4 cl

// default constructor
Circle4::Circle4( int xValue, int yV
   : Point3( xValue, yValue )  // call base-class constructor
{
   setRadius( radiusValue );

} // end Circle4 constructor

// set radius
void Circle4::setRadius( double radiusValue )
{
   radius = ( radiusValue < 0.0 ? 0.0 : radiusValue );

} // end function setRadius

```

Base-class initializer syntax passes arguments to base class **Point3**.



54

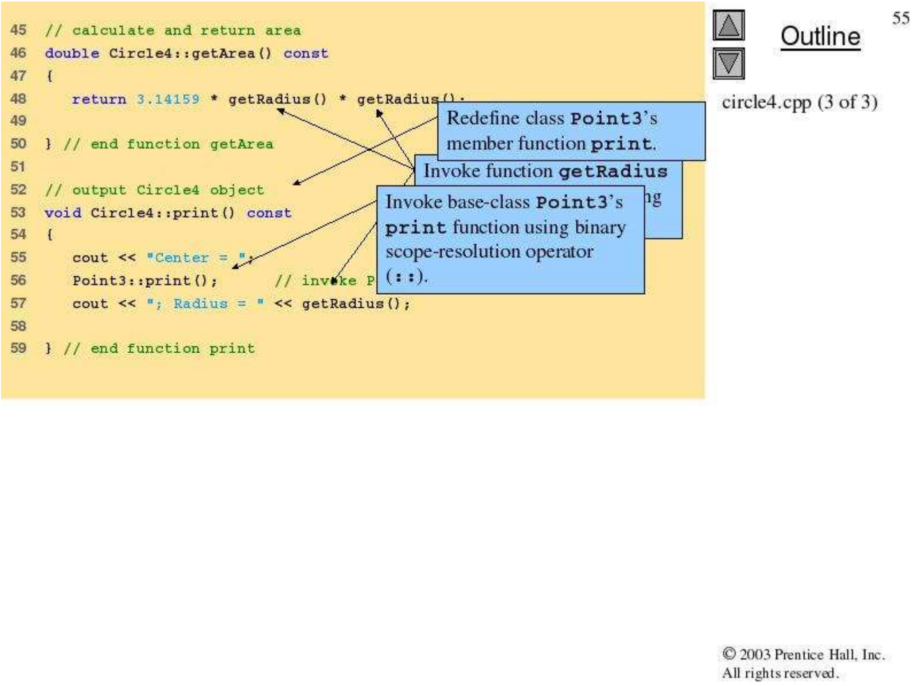
Outline

circle4.cpp (2 of 3)

```
// return radius
double Circle4::getRadius() const
{
   return radius;

} // end function getRadius

// calculate and return diameter
double Circle4::getDiameter() const
{
   return 2 * getRadius();

} // end function getDiameter

// calculate and return circumference
double Circle4::getCircumference() const
{
   return 3.14159 * getDiameter();

} // end function getCircumference

```

Invoke function **getRadius** rather than directly accessing data member **radius**.

Figure 4.23: **Circle4** class that inherits from class **Point3**, which does not provide **protected** data. (part 1 of 2)

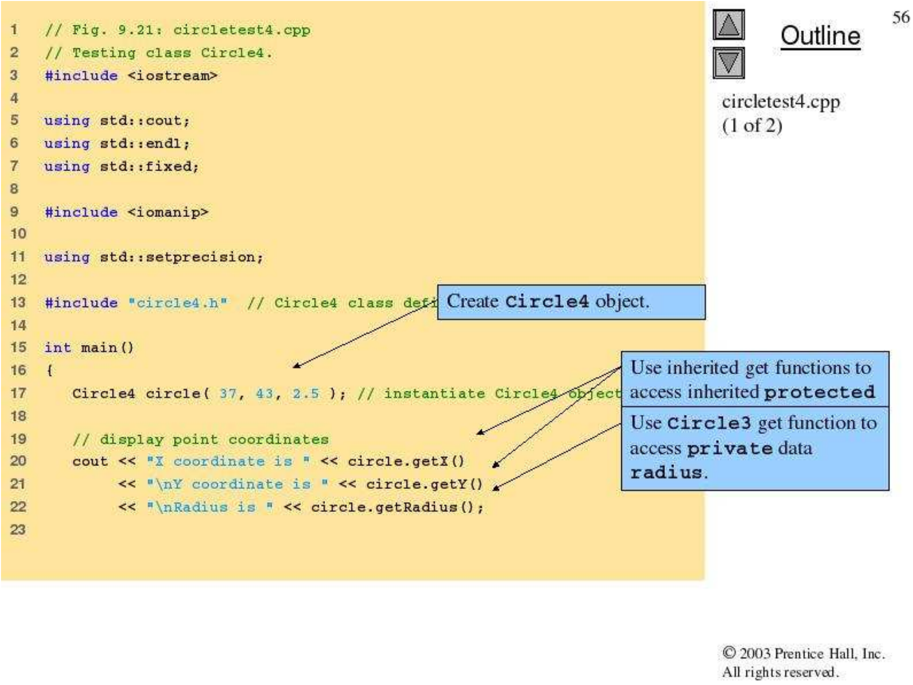


```
// calculate and return area
double Circle4::getArea() const
{
   return 3.14159 * getRadius() * getRadius();

} // end function getArea

// output Circle4 object
void Circle4::print() const
{
   cout << "Center = ";
   Point3::print();      // invoke P
   cout << "; Radius = " << getRadius();

} // end function print
```

Redefine class **Point3**'s member function **print**.

Invoke function **getRadius** ng

Invoke base-class **Point3**'s **print** function using binary scope-resolution operator (**::**).



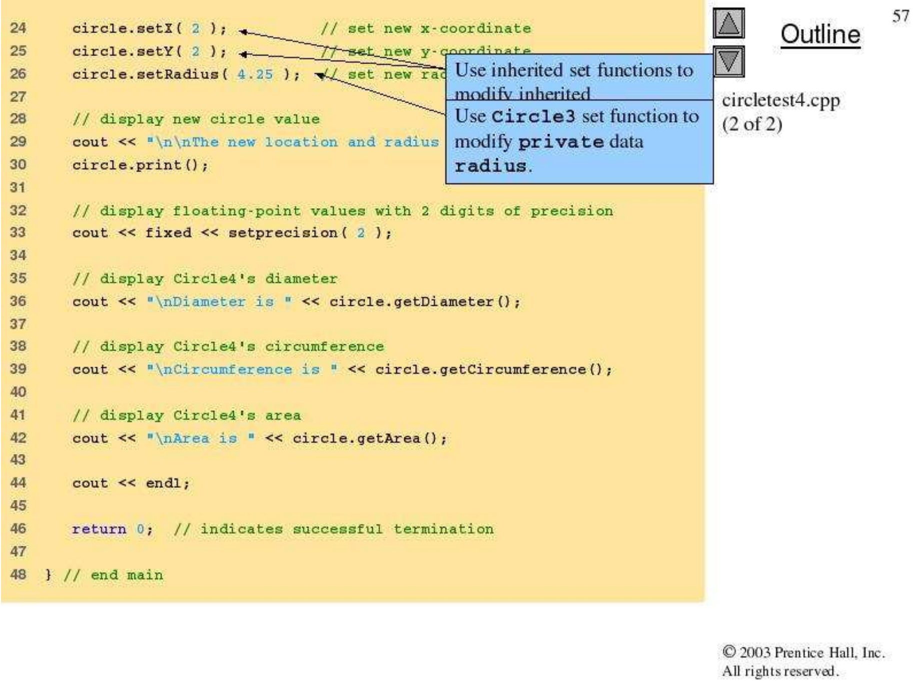
```
// Fig. 9.21: circletest4.cpp
// Testing class Circle4.
#include <iostream>

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

#include <iomanip>

using std::setprecision;

#include "circle4.h"  // Circle4 class defi

int main()
{
   Circle4 circle( 37, 43, 2.5 ); // instantiate Circle4 object

   // display point coordinates
   cout << "X coordinate is " << circle.getX()
        << "\nY coordinate is " << circle.getY()
        << "\nRadius is " << circle.getRadius();

```

Create **Circle4** object.

Use inherited get functions to access inherited **protected**

Use **Circle3** get function to access **private** data **radius**.

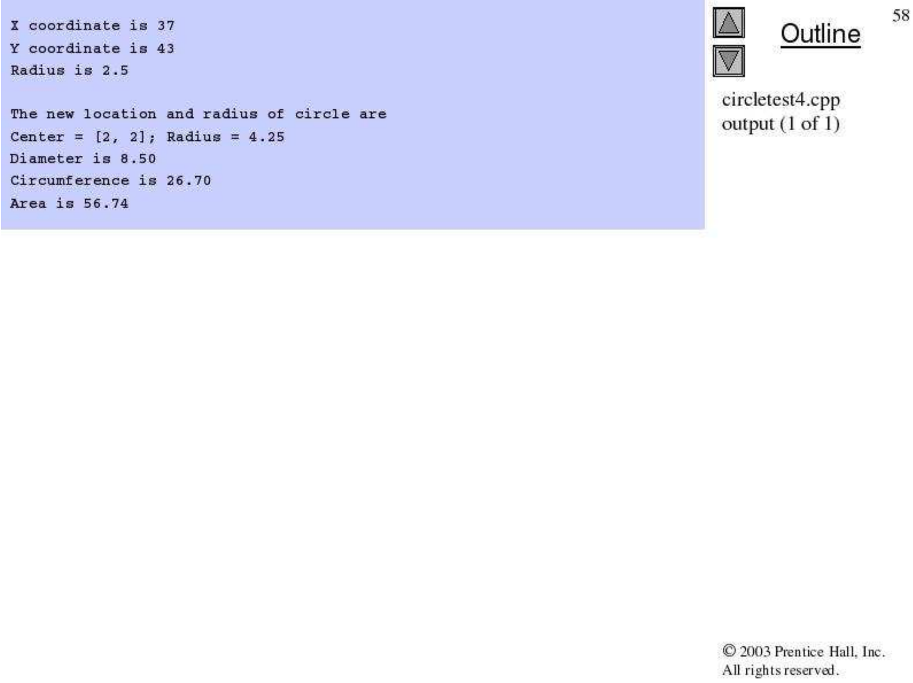
Figure 4.24: **Circle4** class that inherits from class **Point3**, which does not provide **protected** data. (part 2 of 2)



```
   circle.setX( 2 );          // set new x-coordinate
   circle.setY( 2 );          // set new y-coordinate
   circle.setRadius( 4.25 );  // set new rac

   // display new circle value
   cout << "\n\nThe new location and radius
   circle.print();

   // display floating-point values with 2 digits of precision
   cout << fixed << setprecision( 2 );

   // display Circle4's diameter
   cout << "\nDiameter is " << circle.getDiameter();

   // display Circle4's circumference
   cout << "\nCircumference is " << circle.getCircumference();

   // display Circle4's area
   cout << "\nArea is " << circle.getArea();

   cout << endl;

   return 0;  // indicates successful termination

} // end main
```

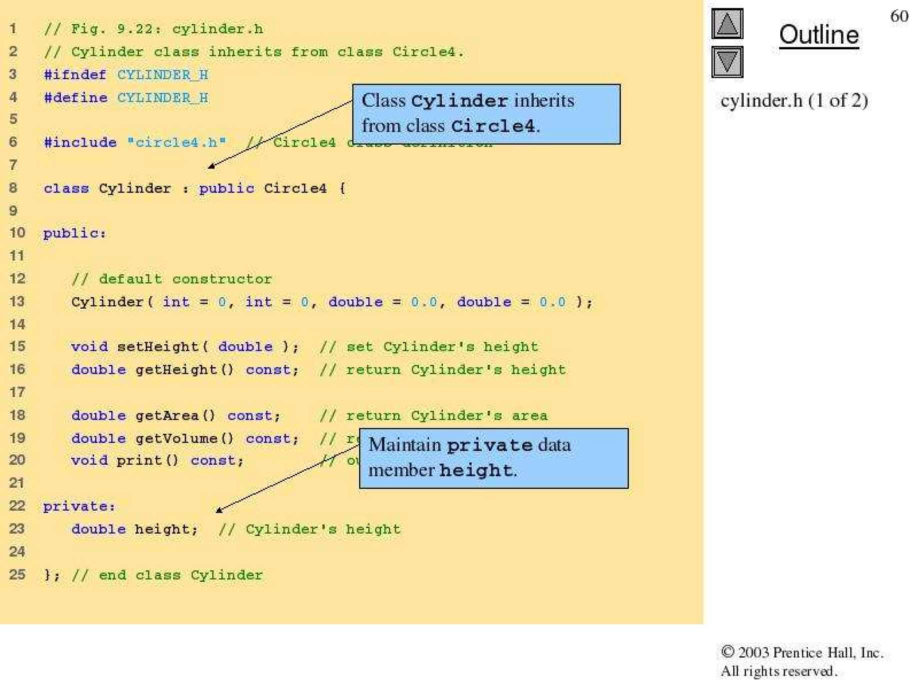
Use inherited set functions to modify inherited

Use **Circle3** set function to modify **private** data **radius**.



```
X coordinate is 37
Y coordinate is 43
Radius is 2.5

The new location and radius of circle are
Center = [2, 2]; Radius = 4.25
Diameter is 8.50
Circumference is 26.70
Area is 56.74
```

Figure 4.25: Base class **private** data is accessible to a derived class via **public** or **protected** member function inherited by the derived class.



60

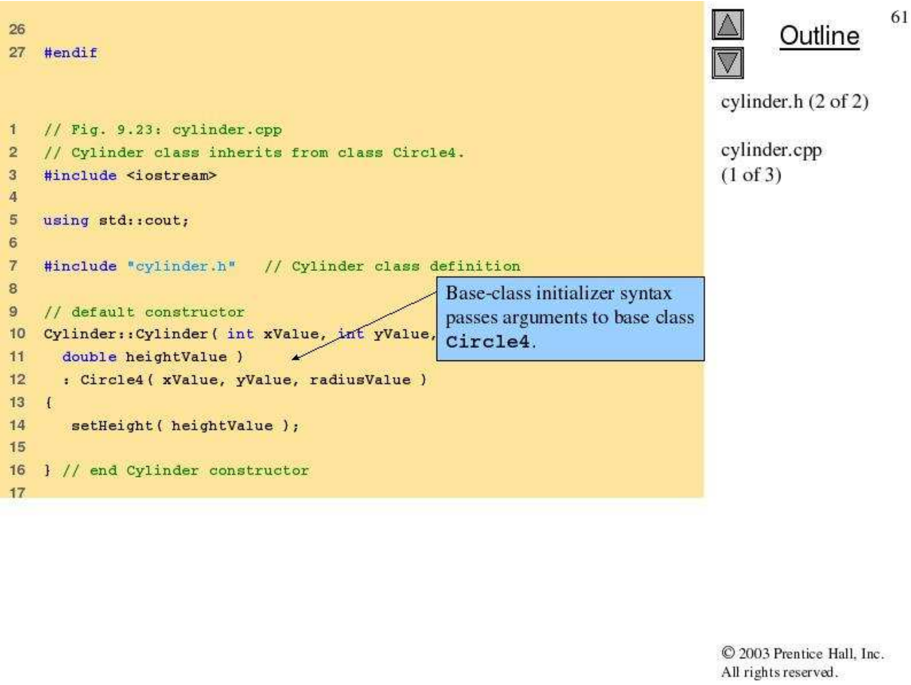
Outline

cylinder.h (1 of 2)

```
// Fig. 9.22: cylinder.h
// Cylinder class inherits from class Circle4.
#ifndef CYLINDER_H
#define CYLINDER_H

#include "circle4.h"  // Circle4

class Cylinder : public Circle4 {

public:

   // default constructor
   Cylinder( int = 0, int = 0, double = 0.0, double = 0.0 );

   void setHeight( double );  // set Cylinder's height
   double getHeight() const;  // return Cylinder's height

   double getArea() const;    // return Cylinder's area
   double getVolume() const;  // r
   void print() const;        // o

private:
   double height;  // Cylinder's height

}; // end class Cylinder
```

Class **Cylinder** inherits from class **Circle4**.

Maintain **private** data member **height**.



61

Outline

cylinder.h (2 of 2)

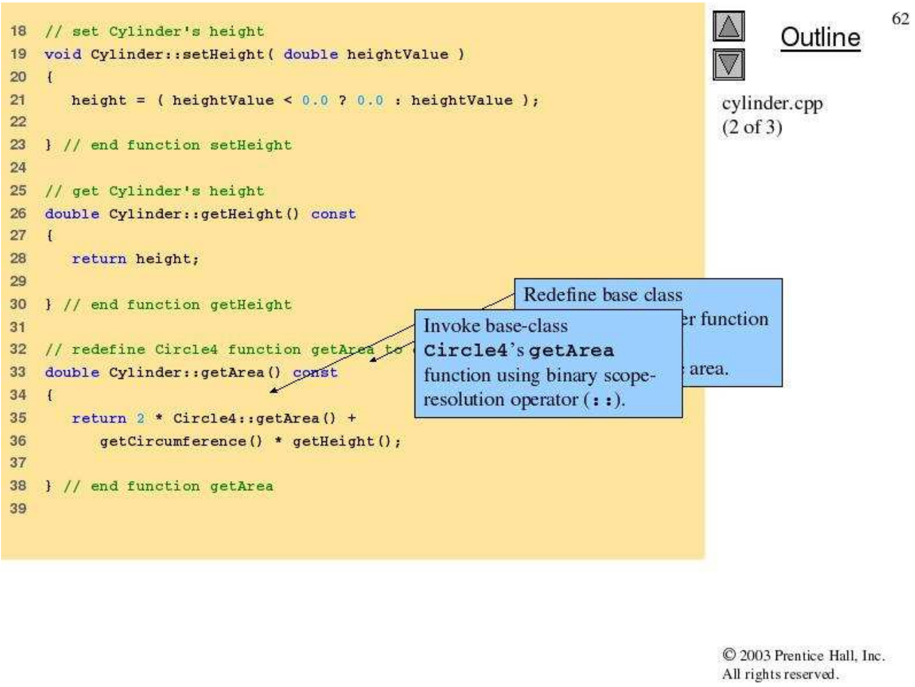
cylinder.cpp
(1 of 3)

```

#endif
```

```
// Fig. 9.23: cylinder.cpp
// Cylinder class inherits from class Circle4.
#include <iostream>

using std::cout;

#include "cylinder.h"   // Cylinder class definition

// default constructor
Cylinder::Cylinder( int xValue, int yValue,
  double heightValue )
  : Circle4( xValue, yValue, radiusValue )
{
   setHeight( heightValue );

} // end Cylinder constructor

```

Base-class initializer syntax passes arguments to base class **Circle4**.

Figure 4.26: **Cylinder** class header file.



62

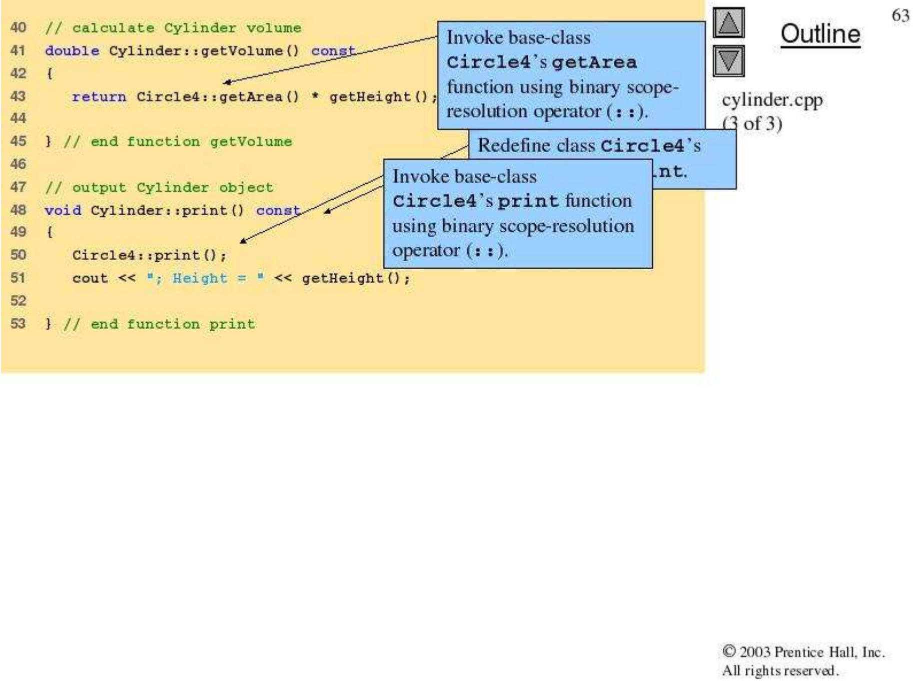
Outline

cylinder.cpp
(2 of 3)

```
// set Cylinder's height
void Cylinder::setHeight( double heightValue )
{
   height = ( heightValue < 0.0 ? 0.0 : heightValue );

} // end function setHeight

// get Cylinder's height
double Cylinder::getHeight() const
{
   return height;

} // end function getHeight

// redefine Circle4 function getArea to
double Cylinder::getArea() const
{
   return 2 * Circle4::getArea() +
      getCircumference() * getHeight();

} // end function getArea

```

Redefine base class
r function
area.

Invoke base-class **Circle4**'s **getArea** function using binary scope-resolution operator (**::**).



63

Outline

cylinder.cpp
(3 of 3)

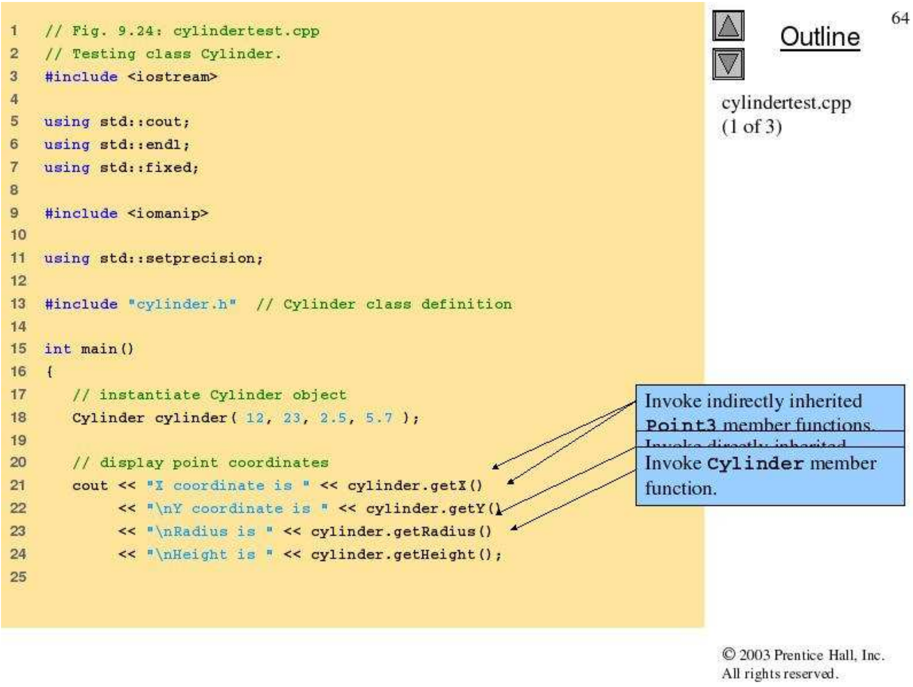
```
// calculate Cylinder volume
double Cylinder::getVolume() const
{
   return Circle4::getArea() * getHeight();

} // end function getVolume

// output Cylinder object
void Cylinder::print() const
{
   Circle4::print();
   cout << "; Height = " << getHeight();

} // end function print
```

Invoke base-class **Circle4**'s **getArea** function using binary scope-resolution operator (**::**).

Redefine class **Circle4**'s
nt.

Invoke base-class **Circle4**'s **print** function using binary scope-resolution operator (**::**).

Figure 4.27: **Cylinder** class inherits from class **Circle4** and redefines member function **getArea**.



64

Outline

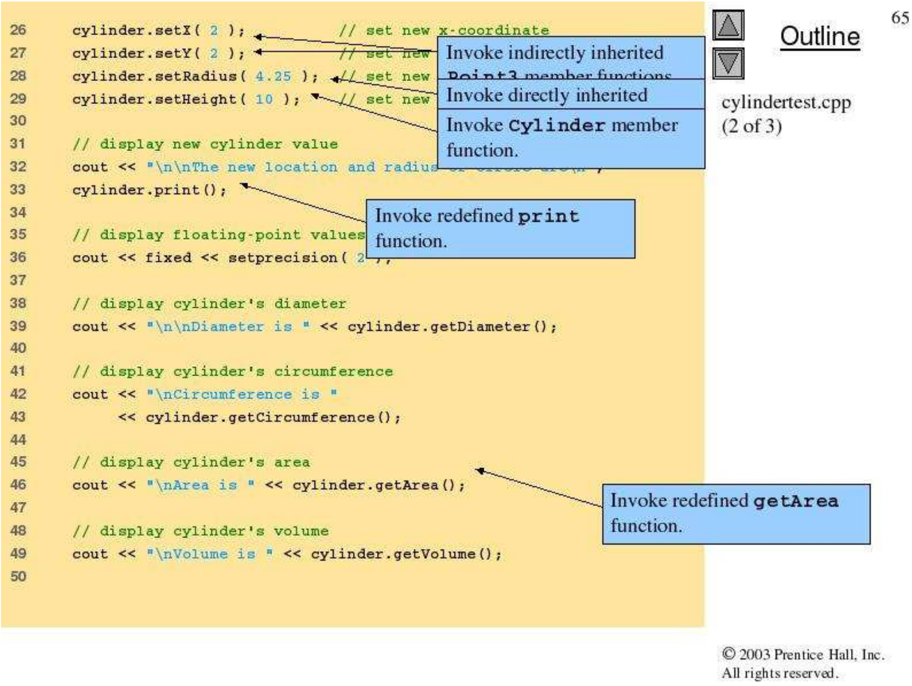
cylindertest.cpp
(1 of 3)

```
// Fig. 9.24: cylindertest.cpp
// Testing class Cylinder.
#include <iostream>

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

#include <iomanip>

using std::setprecision;

#include "cylinder.h"  // Cylinder class definition

int main()
{
   // instantiate Cylinder object
   Cylinder cylinder( 12, 23, 2.5, 5.7 );

   // display point coordinates
   cout << "X coordinate is " << cylinder.getX()
        << "\nY coordinate is " << cylinder.getY()
        << "\nRadius is " << cylinder.getRadius()
        << "\nHeight is " << cylinder.getHeight();

```

Invoke indirectly inherited **Point3** member functions.

Invoke directly inherited

Invoke **Cylinder** member function.



65

Outline

cylindertest.cpp
(2 of 3)

```
   cylinder.setX( 2 );          // set new x-coordinate
   cylinder.setY( 2 );          // set new
   cylinder.setRadius( 4.25 );  // set new
   cylinder.setHeight( 10 );    // set new

   // display new cylinder value
   cout << "\n\nThe new location and radius of circle are\n";
   cylinder.print();

   // display floating-point values
   cout << fixed << setprecision( 2 );

   // display cylinder's diameter
   cout << "\n\nDiameter is " << cylinder.getDiameter();

   // display cylinder's circumference
   cout << "\nCircumference is "
        << cylinder.getCircumference();

   // display cylinder's area
   cout << "\nArea is " << cylinder.getArea();

   // display cylinder's volume
   cout << "\nVolume is " << cylinder.getVolume();

```

Invoke indirectly inherited **Point3** member functions.

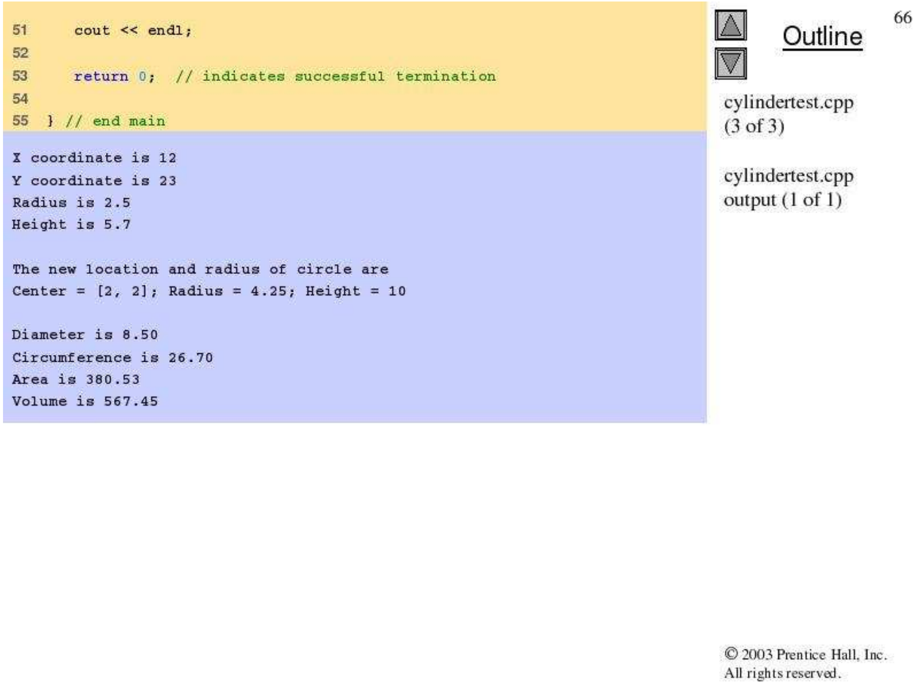
Invoke directly inherited

Invoke **Cylinder** member function.

Invoke redefined **print** function.

Invoke redefined **getArea** function.

Figure 4.28: **Point/Circle/Cylinder** hierarchy test program. (part 1 of 2)



```
   cout << endl;

   return 0;  // indicates successful termination

} // end main
```

```
X coordinate is 12
Y coordinate is 23
Radius is 2.5
Height is 5.7

The new location and radius of circle are
Center = [2, 2]; Radius = 4.25; Height = 10

Diameter is 8.50
Circumference is 26.70
Area is 380.53
Volume is 567.45
```

Figure 4.29: **Point/Circle/Cylinder** hierarchy test program. (part 2 of 2)

4.6 Constructors and Destructors in Derived Classes

- Instantiating derived-class object
  - Chain of constructor calls
    - Derived-class constructor invokes base class constructor
      - Implicitly or explicitly
    - Base of inheritance hierarchy
      - Last constructor called in chain
      - First constructor body to finish executing
      - Example: **Point3/Circle4/Cylinder** hierarchy
        Point3 constructor called last
        Point3 constructor body finishes execution first
    - Initializing data members
      - Each base-class constructor initializes data members
        Inherited by derived class

- Destroying derived-class object
    - Chain of destructor calls
        - Reverse order of constructor chain
        - Destructor of derived-class called first
        - Destructor of next base class up hierarchy next
            - Continue up hierarchy until final base reached; After final base-class destructor, object removed from memory

- Base-class constructors, destructors, assignment operators
    - Not inherited by derived classes
    - Derived class constructors, assignment operators can call
        - Constructors
        - Assignment operators

Next example revisits the point/circle hierarchy by defining class **Point4** (4.30-4.31) and class **Circle5** (4.32-4.34) that contain constructors and destructors, each of which prints a message when it is invoked.

4.7 ”Uses A” and ”Knows A” Relationships

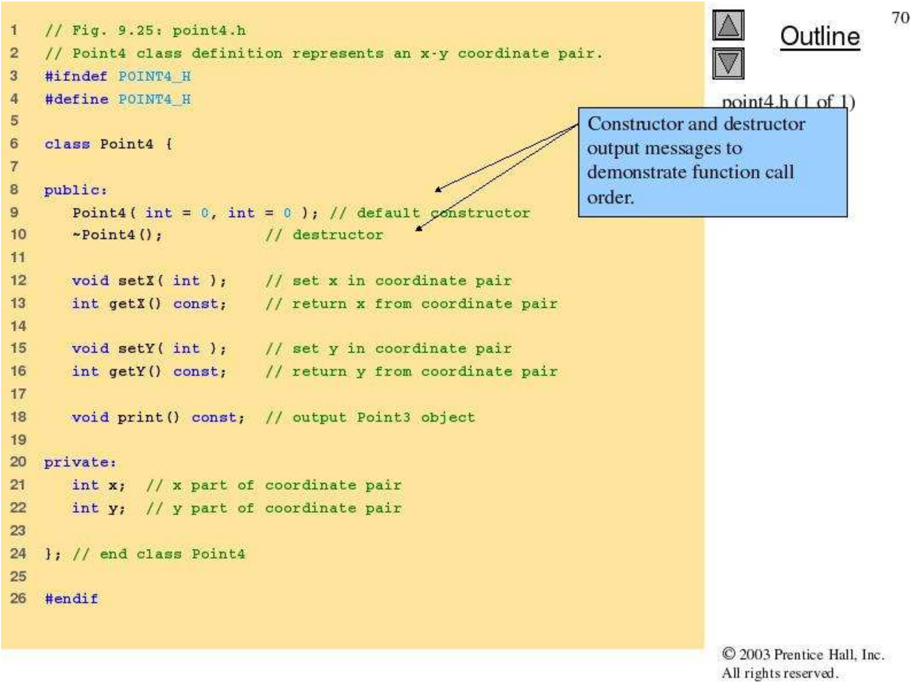
- **”Uses a”**
  - Object uses another object
    - Call non-**private** member function; using pointer, reference or object name
- **”Knows a”** (association)
  - Object aware of another object; contain pointer handle or reference handle
  - Knowledge networks

4.8 public, protected and private Inheritance

4.9 Software Engineering with Inheritance

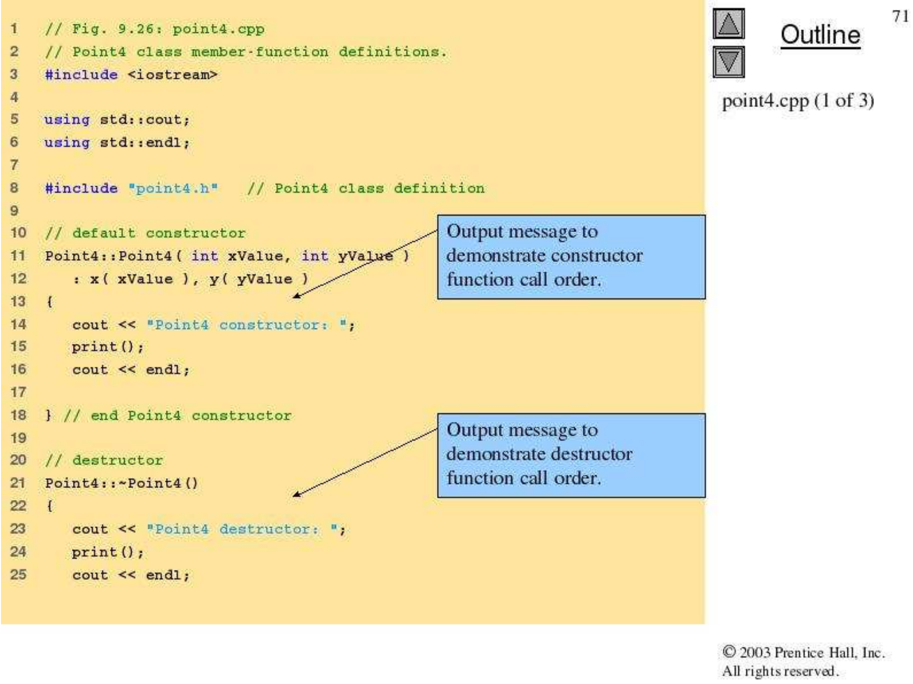
Customizing existing software

- Inherit from existing classes
  - Include additional members
  - Redefine base-class members
  - No direct access to base class’s source code; Link to object code
- Independent software vendors (ISVs)
  - Develop proprietary code for sale/license; available in object-code format
  - Users derive new classes; without accessing ISV proprietary source code



```
// Fig. 9.25: point4.h
// Point4 class definition represents an x-y coordinate pair.
#ifndef POINT4_H
#define POINT4_H

class Point4 {

public:
   Point4( int = 0, int = 0 ); // default constructor
   ~Point4();           // destructor

   void setX( int );    // set x in coordinate pair
   int getX() const;    // return x from coordinate pair

   void setY( int );    // set y in coordinate pair
   int getY() const;    // return y from coordinate pair

   void print() const;  // output Point3 object

private:
   int x;  // x part of coordinate pair
   int y;  // y part of coordinate pair

}; // end class Point4

#endif
```

Constructor and destructor output messages to demonstrate function call order.

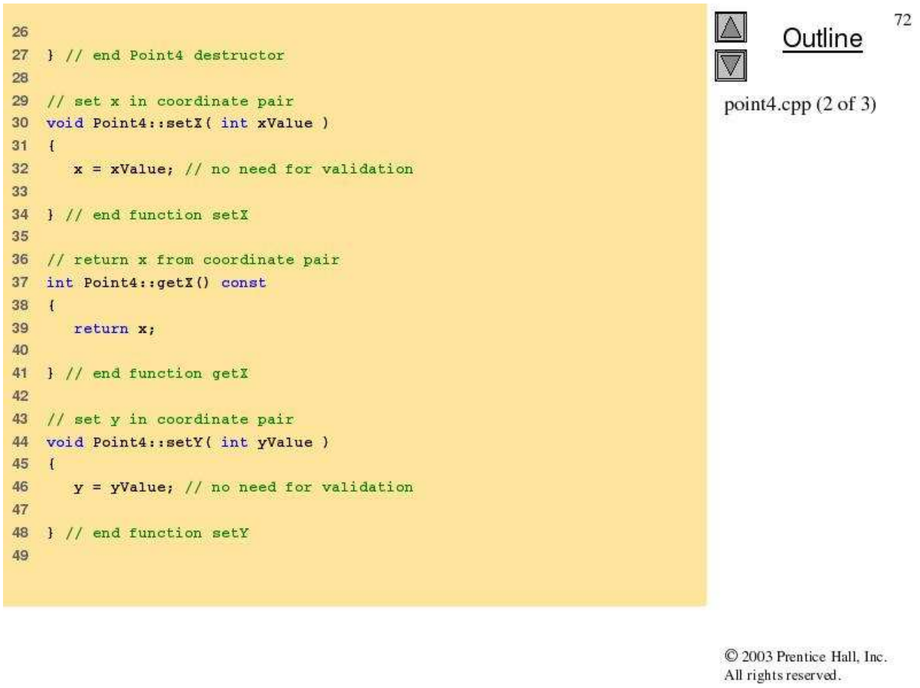


```
// Fig. 9.26: point4.cpp
// Point4 class member-function definitions.
#include <iostream>

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

#include "point4.h"   // Point4 class definition

// default constructor
Point4::Point4( int xValue, int yValue )
   : x( xValue ), y( yValue )
{
   cout << "Point4 constructor: ";
   print();
   cout << endl;

} // end Point4 constructor

// destructor
Point4::~Point4()
{
   cout << "Point4 destructor: ";
   print();
   cout << endl;
```

Output message to demonstrate constructor function call order.

Output message to demonstrate destructor function call order.

Figure 4.30: **Point4** class header file and **Point4** base class contains a constructor and a destructor. (part 1 of 2)

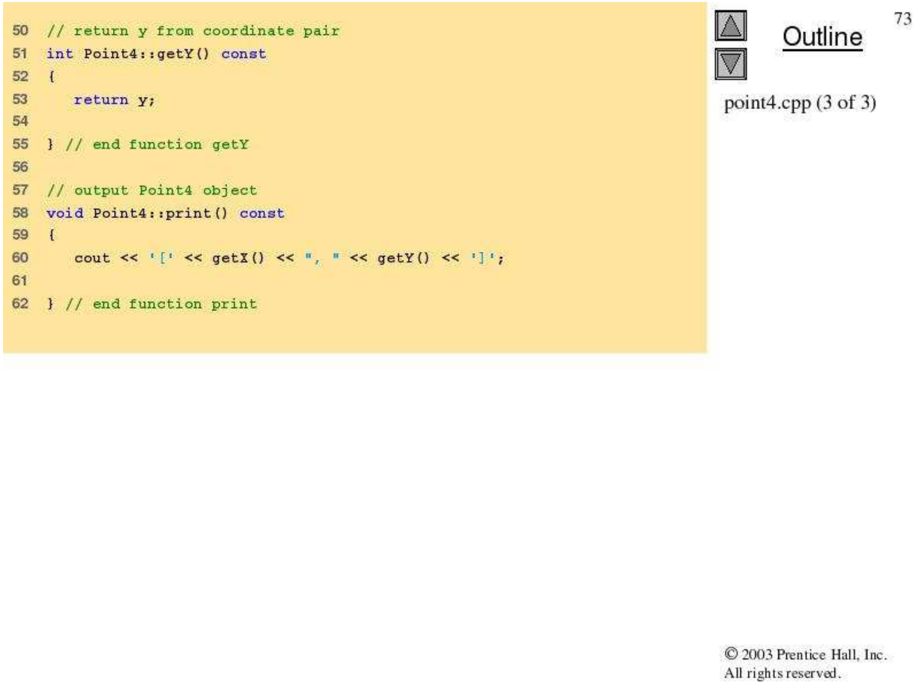


```cpp

} // end Point4 destructor

// set x in coordinate pair
void Point4::setX( int xValue )
{
   x = xValue; // no need for validation

} // end function setX

// return x from coordinate pair
int Point4::getX() const
{
   return x;

} // end function getX

// set y in coordinate pair
void Point4::setY( int yValue )
{
   y = yValue; // no need for validation

} // end function setY

```

72

▲ ▼ Outline

point4.cpp (2 of 3)



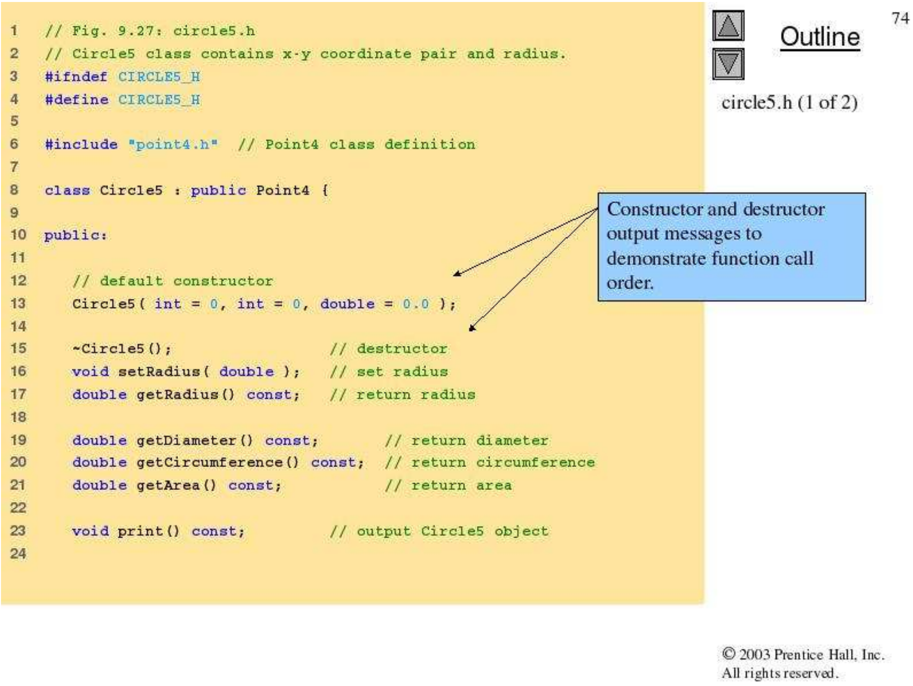
```cpp
// return y from coordinate pair
int Point4::getY() const
{
   return y;

} // end function getY

// output Point4 object
void Point4::print() const
{
   cout << '[' << getX() << ", " << getY() << ']';

} // end function print
```

73

▲ ▼ Outline

point4.cpp (3 of 3)

Figure 4.31: **Point4** base class contains a constructor and a destructor. (part 2of 2)

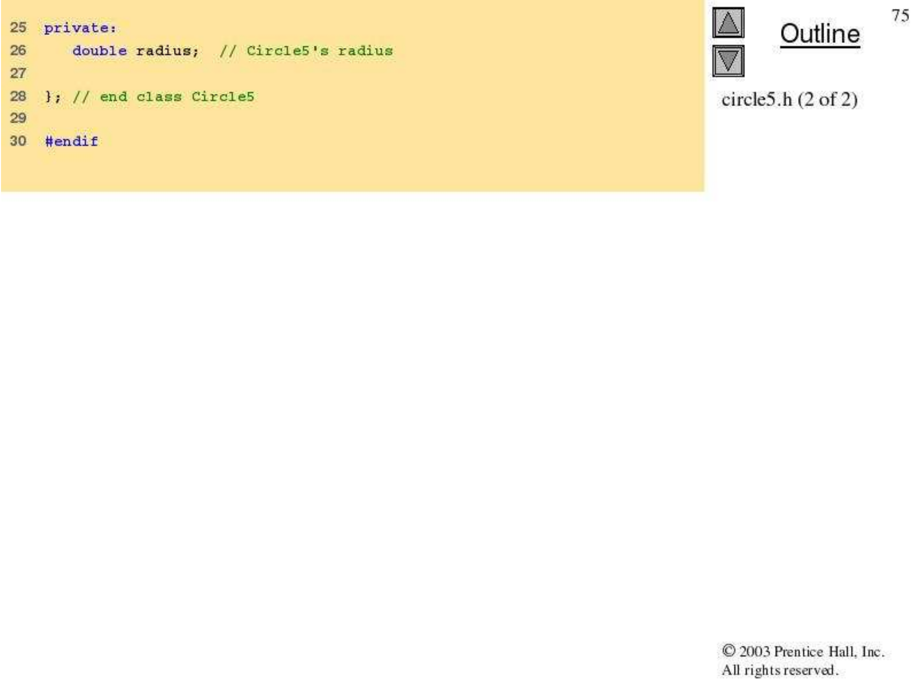


74

circle5.h (1 of 2)

```
// Fig. 9.27: circle5.h
// Circle5 class contains x-y coordinate pair and radius.
#ifndef CIRCLE5_H
#define CIRCLE5_H

#include "point4.h"  // Point4 class definition

class Circle5 : public Point4 {

public:

   // default constructor
   Circle5( int = 0, int = 0, double = 0.0 );

   ~Circle5();                  // destructor
   void setRadius( double );    // set radius
   double getRadius() const;    // return radius

   double getDiameter() const;       // return diameter
   double getCircumference() const;  // return circumference
   double getArea() const;           // return area

   void print() const;          // output Circle5 object

```

Constructor and destructor output messages to demonstrate function call order.

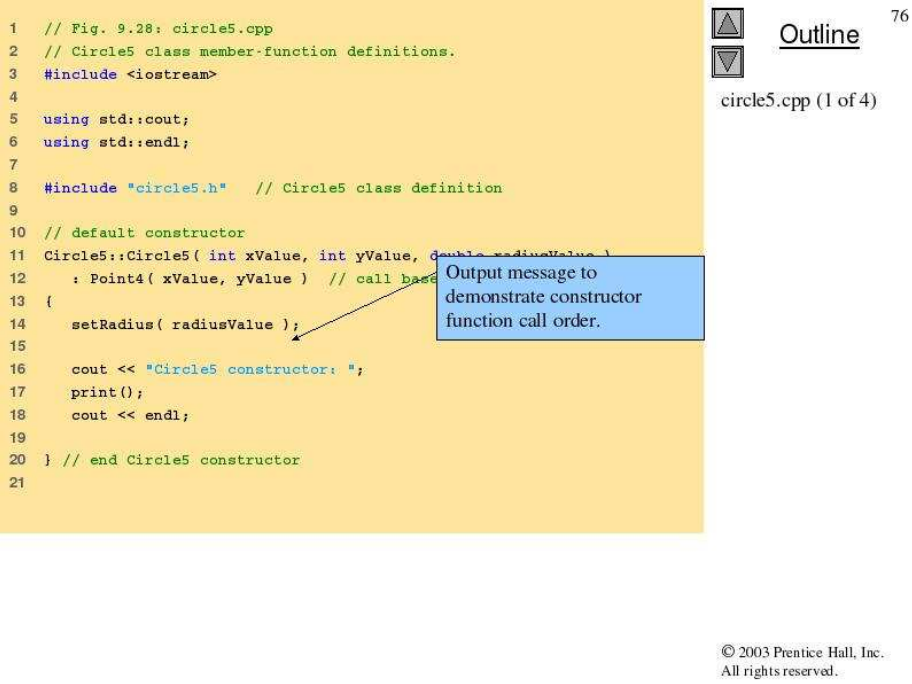


75

circle5.h (2 of 2)

```
private:
   double radius;  // Circle5's radius

}; // end class Circle5

#endif
```

Figure 4.32: **Circle5** class header file.



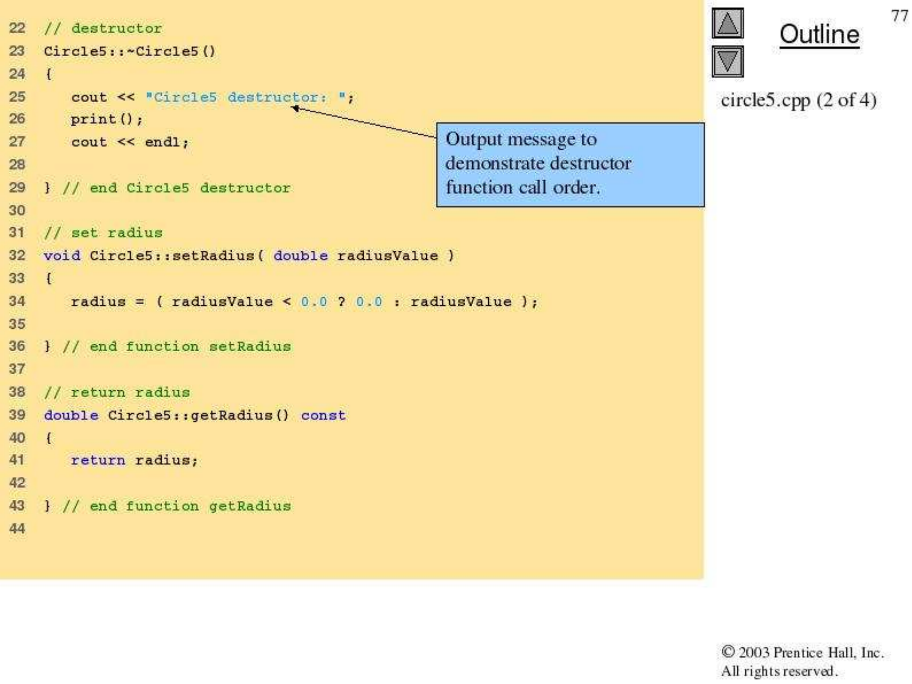
76

Outline

circle5.cpp (1 of 4)

```
// Fig. 9.28: circle5.cpp
// Circle5 class member-function definitions.
#include <iostream>

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

#include "circle5.h"   // Circle5 class definition

// default constructor
Circle5::Circle5( int xValue, int yValue, double radiusValue )
   : Point4( xValue, yValue )  // call base
{
   setRadius( radiusValue );

   cout << "Circle5 constructor: ";
   print();
   cout << endl;

} // end Circle5 constructor

```

Output message to demonstrate constructor function call order.



77

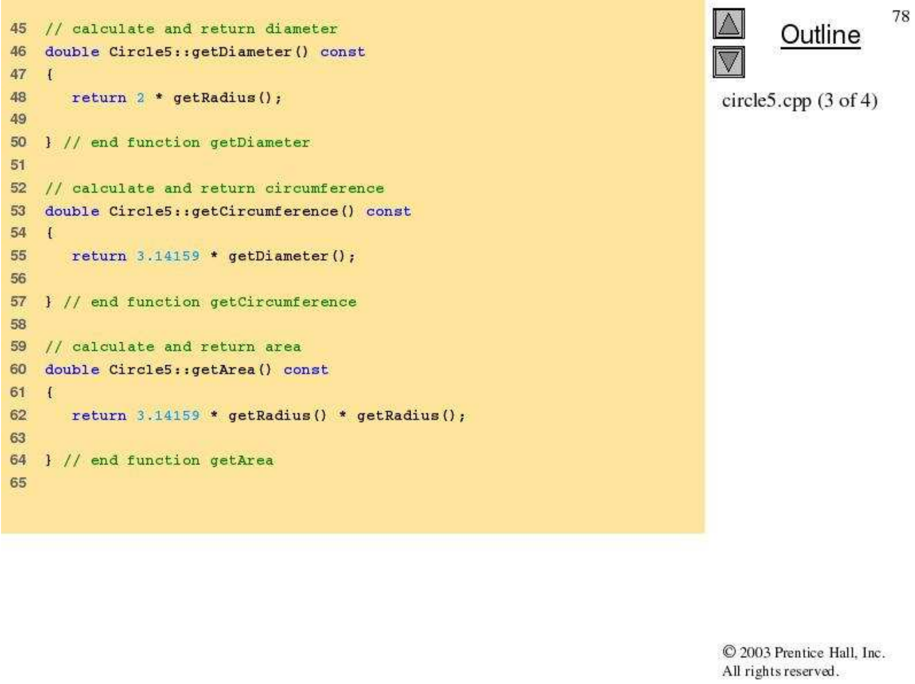
Outline

circle5.cpp (2 of 4)

```
// destructor
Circle5::~Circle5()
{
   cout << "Circle5 destructor: ";
   print();
   cout << endl;

} // end Circle5 destructor

// set radius
void Circle5::setRadius( double radiusValue )
{
   radius = ( radiusValue < 0.0 ? 0.0 : radiusValue );

} // end function setRadius

// return radius
double Circle5::getRadius() const
{
   return radius;

} // end function getRadius

```

Output message to demonstrate destructor function call order.

Figure 4.33: **Circle5** class inherits from class **Point4**. (part 1 of 2)

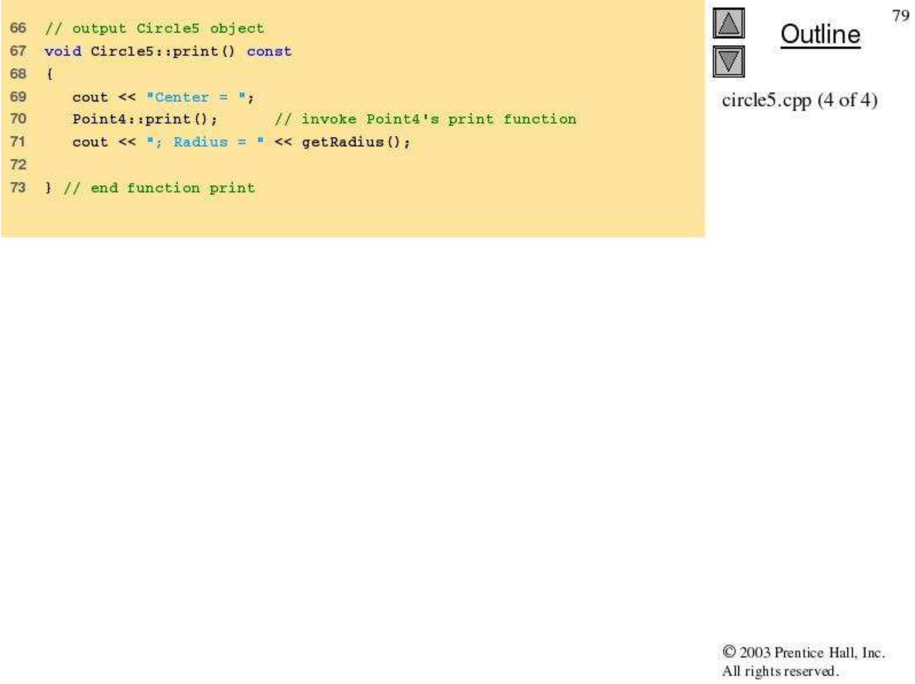


```cpp
// calculate and return diameter
double Circle5::getDiameter() const
{
   return 2 * getRadius();

} // end function getDiameter

// calculate and return circumference
double Circle5::getCircumference() const
{
   return 3.14159 * getDiameter();

} // end function getCircumference

// calculate and return area
double Circle5::getArea() const
{
   return 3.14159 * getRadius() * getRadius();

} // end function getArea

```

78

Outline

circle5.cpp (3 of 4)



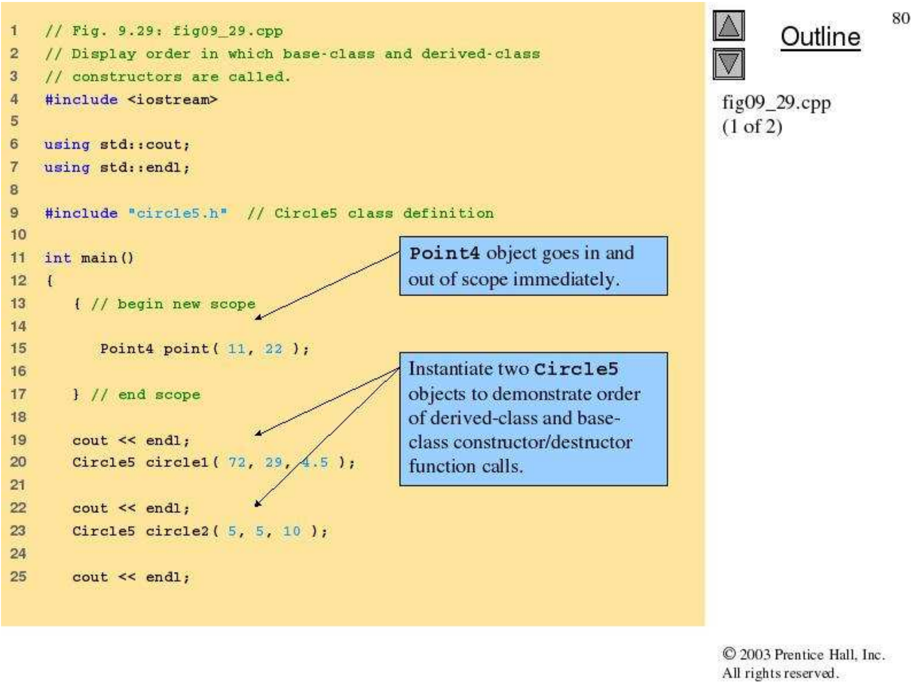
```cpp
// output Circle5 object
void Circle5::print() const
{
   cout << "Center = ";
   Point4::print();      // invoke Point4's print function
   cout << "; Radius = " << getRadius();

} // end function print
```

79

Outline

circle5.cpp (4 of 4)

Figure 4.34: **Circle5** class inherits from class **Point4**. (part 2 of 2)



```
// Fig. 9.29: fig09_29.cpp
// Display order in which base-class and derived-class
// constructors are called.
#include <iostream>

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

#include "circle5.h"  // Circle5 class definition

int main()
{
   { // begin new scope

      Point4 point( 11, 22 );

   } // end scope

   cout << endl;
   Circle5 circle1( 72, 29, 4.5 );

   cout << endl;
   Circle5 circle2( 5, 5, 10 );

   cout << endl;
```

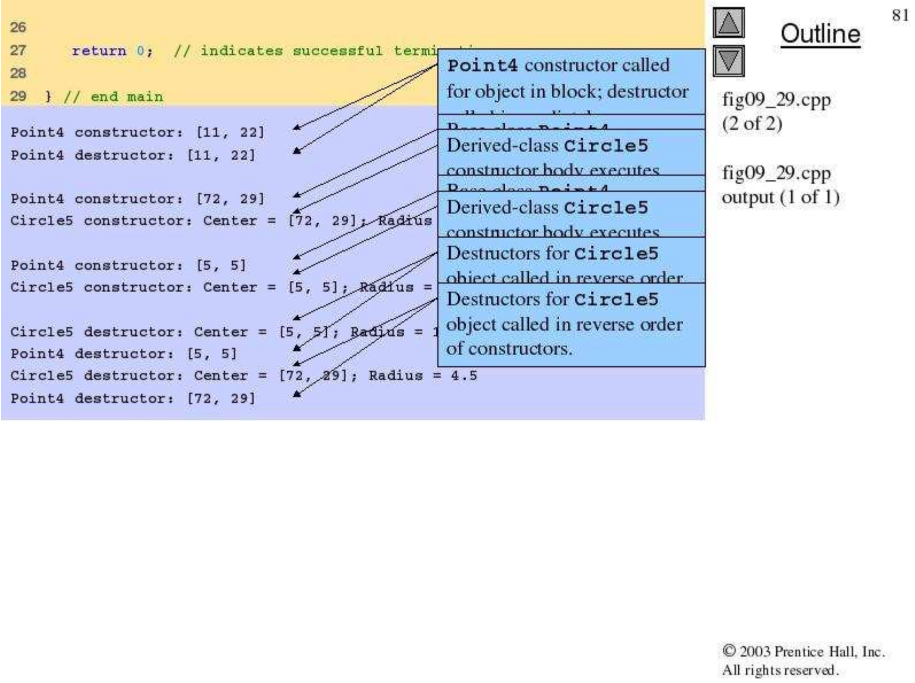
Point4 object goes in and out of scope immediately.

Instantiate two **Circle5** objects to demonstrate order of derived-class and base-class constructor/destructor function calls.

80

Outline

fig09_29.cpp
(1 of 2)



```

   return 0;  // indicates successful termi

} // end main
```

```
Point4 constructor: [11, 22]
Point4 destructor: [11, 22]

Point4 constructor: [72, 29]
Circle5 constructor: Center = [72, 29]; Radius

Point4 constructor: [5, 5]
Circle5 constructor: Center = [5, 5]; Radius =

Circle5 destructor: Center = [5, 5]; Radius = 1
Point4 destructor: [5, 5]
Circle5 destructor: Center = [72, 29]; Radius = 4.5
Point4 destructor: [72, 29]
```

Point4 constructor called for object in block; destructor

Derived-class **Circle5** constructor body executes

Derived-class **Circle5** constructor body executes

Destructors for **Circle5** object called in reverse order

Destructors for **Circle5** object called in reverse order of constructors.

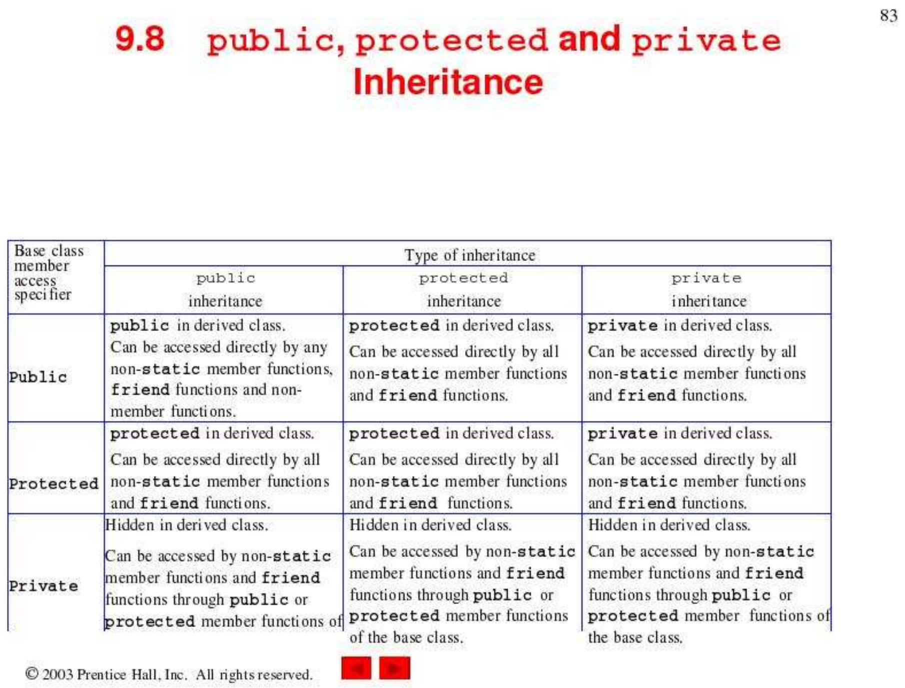
81

Outline

fig09_29.cpp
(2 of 2)

fig09_29.cpp
output (1 of 1)

Figure 4.35: Constructor and destructor call order.



83

9.8 public, protected and private Inheritance

Base class member access specifier	Type of inheritance		
	public inheritance	protected inheritance	private inheritance
Public	**public** in derived class. Can be accessed directly by any non-**static** member functions, **friend** functions and non-member functions.	**protected** in derived class. Can be accessed directly by all non-**static** member functions and **friend** functions.	**private** in derived class. Can be accessed directly by all non-**static** member functions and **friend** functions.
Protected	**protected** in derived class. Can be accessed directly by all non-**static** member functions and **friend** functions.	**protected** in derived class. Can be accessed directly by all non-**static** member functions and **friend** functions.	**private** in derived class. Can be accessed directly by all non-**static** member functions and **friend** functions.
Private	Hidden in derived class. Can be accessed by non-**static** member functions and **friend** functions through **public** or **protected** member functions of	Hidden in derived class. Can be accessed by non-**static** member functions and **friend** functions through **public** or **protected** member functions of the base class.	Hidden in derived class. Can be accessed by non-**static** member functions and **friend** functions through **public** or **protected** member functions of the base class.

Figure 4.36: Summary of base–class member accessibility in a derived class.

Chapter 5

Object-Oriented Programming: Polymorphism

5.1 Introduction

- Polymorphism
  - ”Program in the general”
  - Treat objects in same class hierarchy as if all base class
  - Virtual functions and dynamic binding; will explain how polymorphism works
  - Makes programs extensible; new classes added easily, can still be processed
- In our examples
  - Use abstract base class **Shape**
    - Defines common interface (functionality)
    - **Point**, **Circle** and **Cylinder** inherit from **Shape**
  - Class **Employee** for a natural example

5.2 Relationships Among Objects in an Inheritance Hierarchy

- Previously (Section 9.4),
  - **Circle** inherited from **Point**

  - Manipulated **Point** and **Circle** objects using member functions

- Now
  - Invoke functions using base-class/derived-class pointers
  - Introduce **virtual** functions

- Key concept
  - Derived-class object can be treated as base-class object
    - ”is-a” relationship
    - Base class is not a derived class object

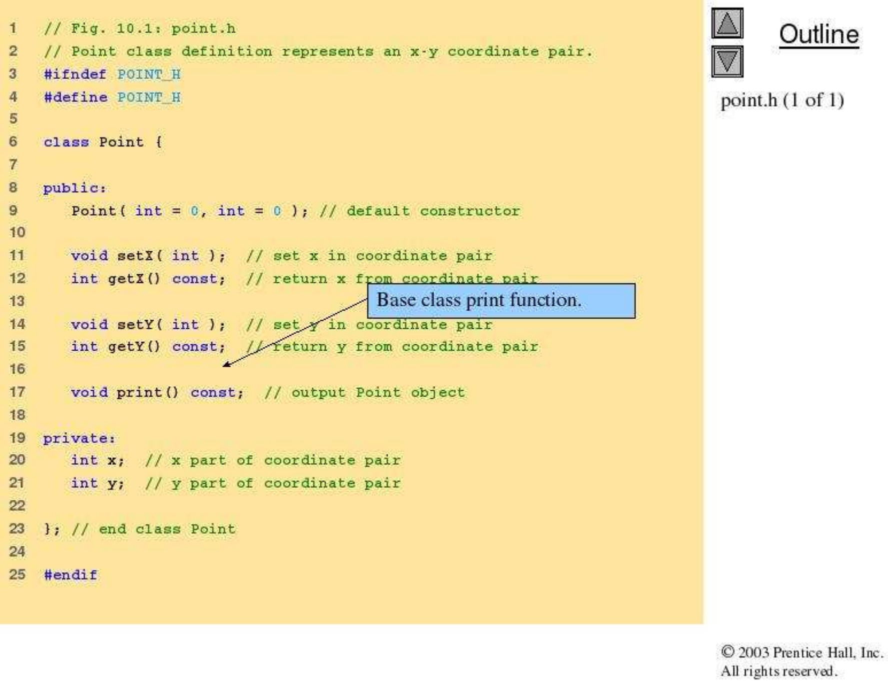
5.2.1 Invoking Base-Class Functions from Derived-Class Objects

Aim pointers (base, derived) at objects (base, derived)

- Base pointer aimed at base object
- Derived pointer aimed at derived object; both straightforward
- Base pointer aimed at derived object
  - ”is a” relationship; **Circle** ”is a” **Point**
  - Will invoke base class functions
- Function call depends on the class of the pointer/handle
  - Does not depend on object to which it points
  - With **virtual** functions, this can be changed (more later)

5.2.2 Aiming Derived-Class Pointers at Base-Class Objects

- Previous example
  - Aimed base-class pointer at derived object; **Circle** ”is a” **Point**
- Aim a derived-class pointer at a base-class object
  - Compiler error



```cpp
// Fig. 10.1: point.h
// Point class definition represents an x-y coordinate pair.
#ifndef POINT_H
#define POINT_H

class Point {

public:
   Point( int = 0, int = 0 ); // default constructor

   void setX( int );  // set x in coordinate pair
   int getX() const;  // return x from coordinate pair

   void setY( int );  // set y in coordinate pair
   int getY() const;  // return y from coordinate pair

   void print() const;  // output Point object

private:
   int x;  // x part of coordinate pair
   int y;  // y part of coordinate pair

}; // end class Point

#endif
```

Base class print function.

5

Outline

point.h (1 of 1)

© 2003 Prentice Hall, Inc.
All rights reserved.

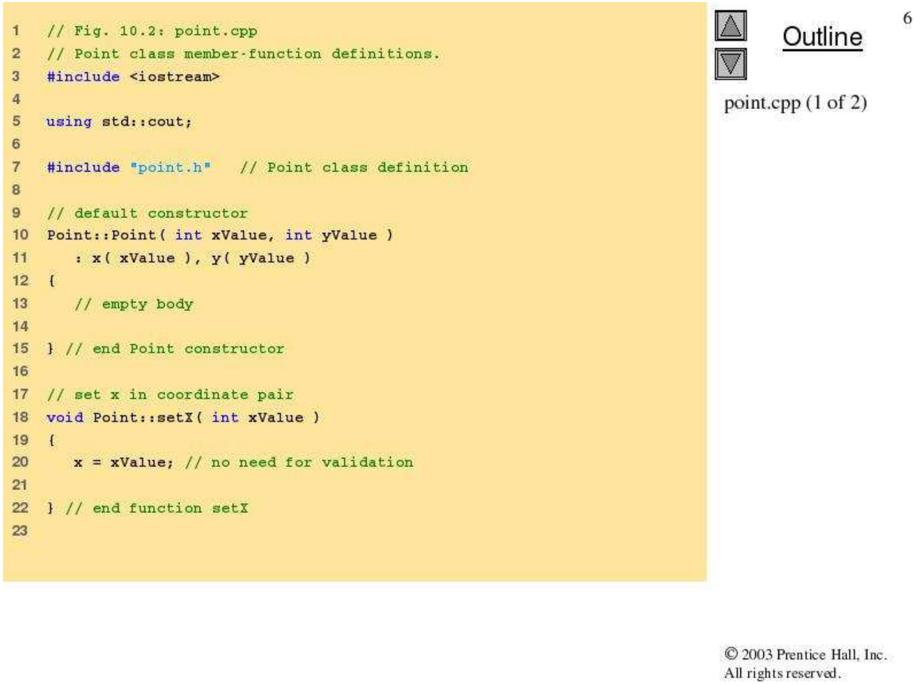
Figure 5.1: **Point** class header file.

- No ”is a” relationship
- **Point** is not a **Circle**
- **Circle** has data/functions that **Point** does not
  - **setRadius** (defined in **Circle**) not defined in **Point**

- Can cast base-object”s address to derived-class pointer
  - Called downcasting (more in 10.9)
  - Allows derived-class functionality

5.2.3 Derived-Class Member-Function Calls via Base-Class Pointers

- Handle (pointer/reference)
  - Base-pointer can aim at derived-object; but can only call base-class functions
  - Calling derived-class functions is a compiler error; functions not defined in base-class

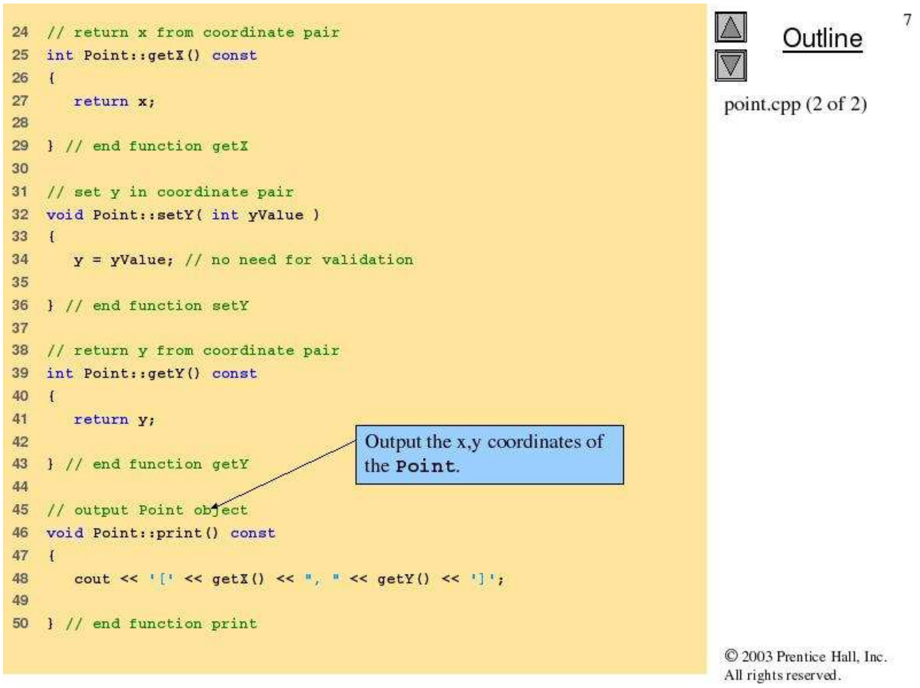


```
// Fig. 10.2: point.cpp
// Point class member-function definitions.
#include <iostream>

using std::cout;

#include "point.h"   // Point class definition

// default constructor
Point::Point( int xValue, int yValue )
   : x( xValue ), y( yValue )
{
   // empty body

} // end Point constructor

// set x in coordinate pair
void Point::setX( int xValue )
{
   x = xValue; // no need for validation

} // end function setX

```

6

Outline

point.cpp (1 of 2)



```
// return x from coordinate pair
int Point::getX() const
{
   return x;

} // end function getX

// set y in coordinate pair
void Point::setY( int yValue )
{
   y = yValue; // no need for validation

} // end function setY

// return y from coordinate pair
int Point::getY() const
{
   return y;

} // end function getY

// output Point object
void Point::print() const
{
   cout << '[' << getX() << ", " << getY() << ']';

} // end function print
```

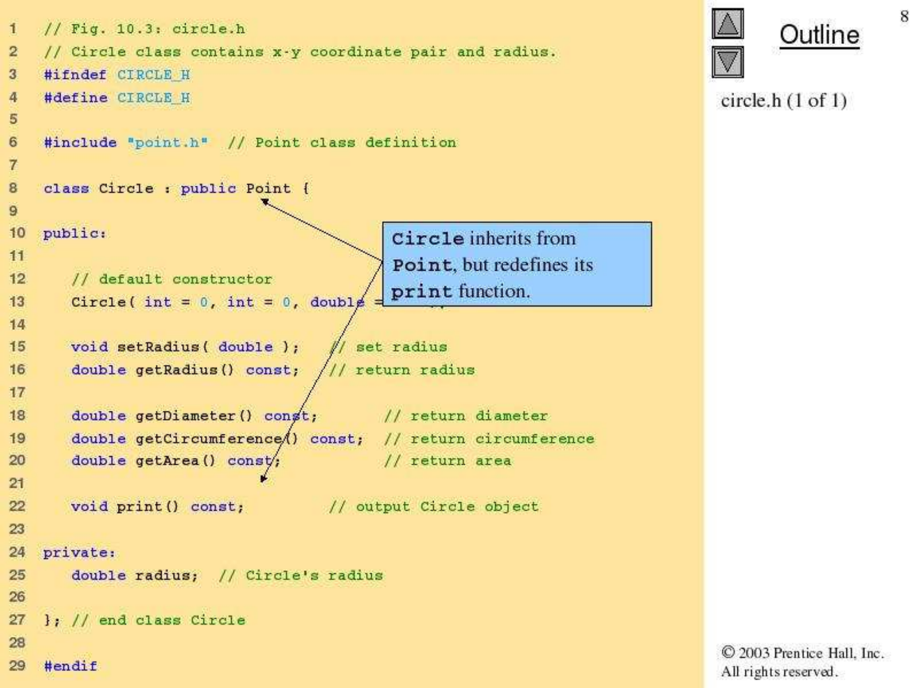
Output the x,y coordinates of the **Point**.

7

Outline

point.cpp (2 of 2)

Figure 5.2: **Point** class represents an xy-coordinate pair.



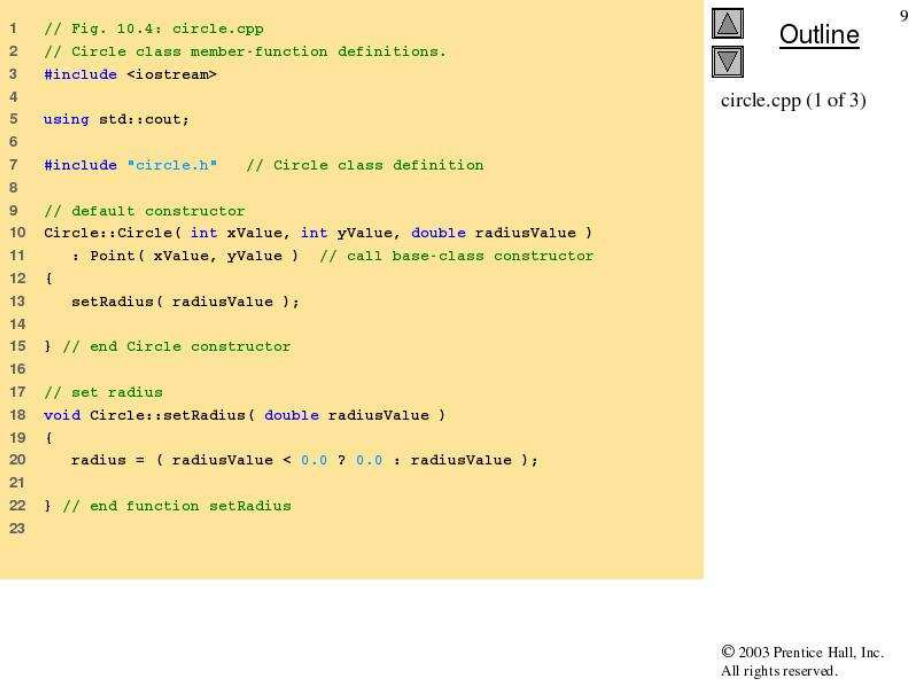
```
// Fig. 10.3: circle.h
// Circle class contains x-y coordinate pair and radius.
#ifndef CIRCLE_H
#define CIRCLE_H

#include "point.h"  // Point class definition

class Circle : public Point {

public:

   // default constructor
   Circle( int = 0, int = 0, double =

   void setRadius( double );     // set radius
   double getRadius() const;     // return radius

   double getDiameter() const;       // return diameter
   double getCircumference() const;  // return circumference
   double getArea() const;           // return area

   void print() const;         // output Circle object

private:
   double radius;  // Circle's radius

}; // end class Circle

#endif
```

Circle inherits from **Point**, but redefines its **print** function.

8

Outline

circle.h (1 of 1)



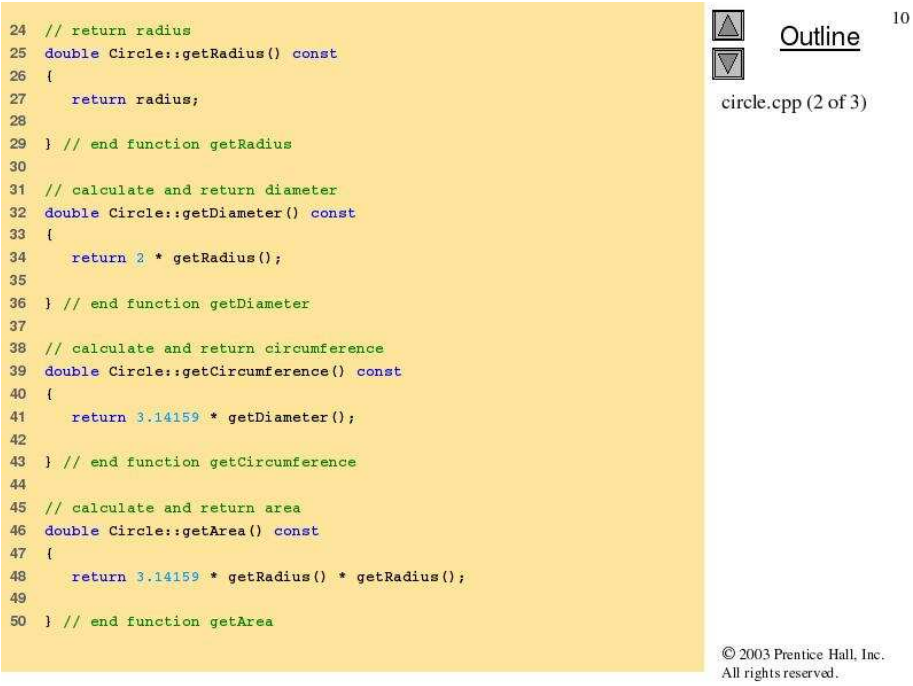
```
// Fig. 10.4: circle.cpp
// Circle class member-function definitions.
#include <iostream>

using std::cout;

#include "circle.h"   // Circle class definition

// default constructor
Circle::Circle( int xValue, int yValue, double radiusValue )
   : Point( xValue, yValue )  // call base-class constructor
{
   setRadius( radiusValue );

} // end Circle constructor

// set radius
void Circle::setRadius( double radiusValue )
{
   radius = ( radiusValue < 0.0 ? 0.0 : radiusValue );

} // end function setRadius

```

9

Outline

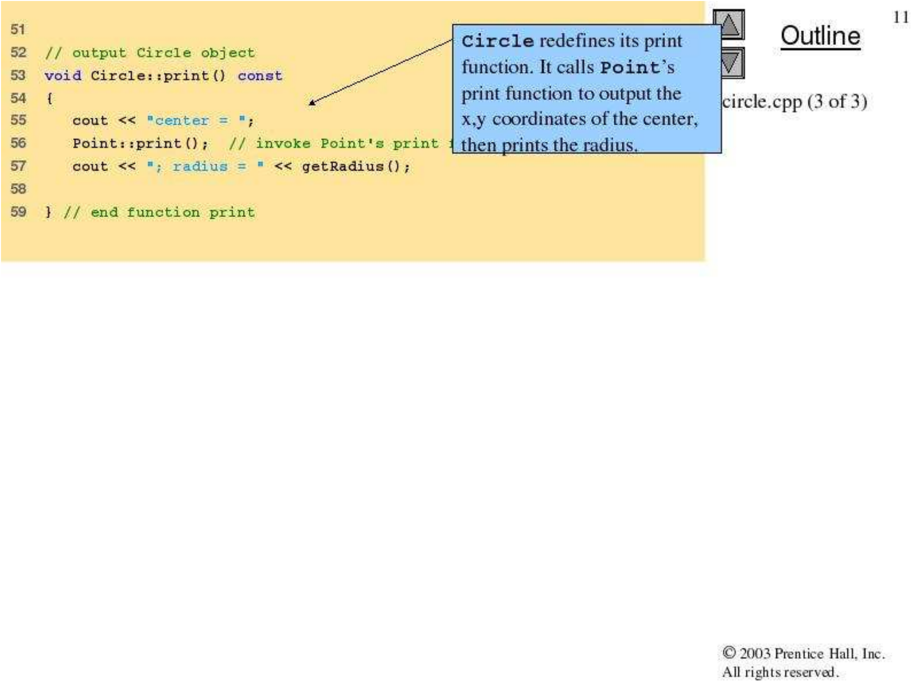
circle.cpp (1 of 3)

Figure 5.3: **Circle** class header file.



```
// return radius
double Circle::getRadius() const
{
   return radius;

} // end function getRadius

// calculate and return diameter
double Circle::getDiameter() const
{
   return 2 * getRadius();

} // end function getDiameter

// calculate and return circumference
double Circle::getCircumference() const
{
   return 3.14159 * getDiameter();

} // end function getCircumference

// calculate and return area
double Circle::getArea() const
{
   return 3.14159 * getRadius() * getRadius();

} // end function getArea
```

circle.cpp (2 of 3)

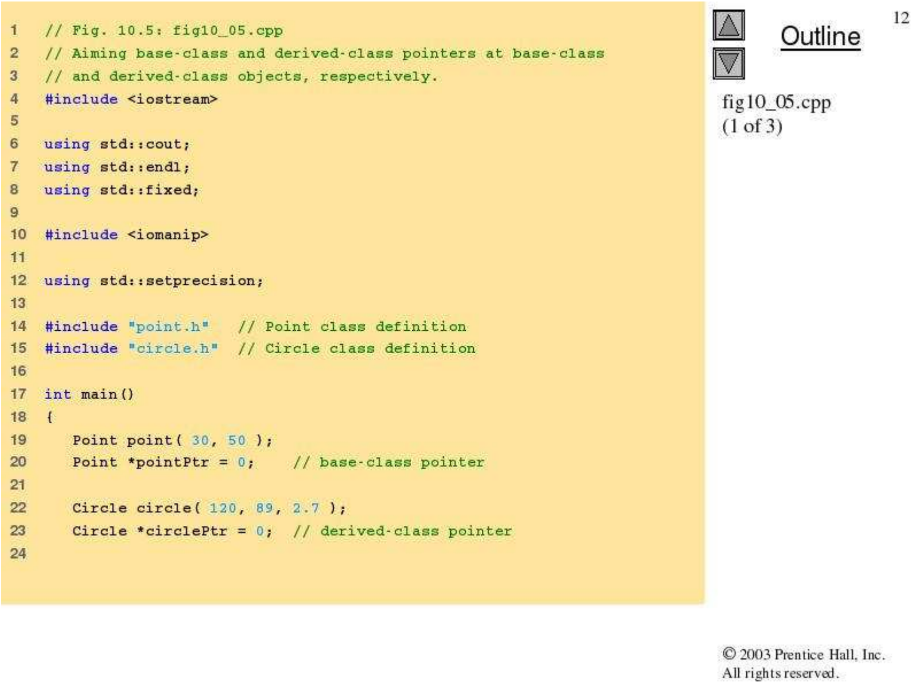


```

// output Circle object
void Circle::print() const
{
   cout << "center = ";
   Point::print();  // invoke Point's print f
   cout << "; radius = " << getRadius();

} // end function print
```

Circle redefines its print function. It calls **Point**'s print function to output the x,y coordinates of the center, then prints the radius.

circle.cpp (3 of 3)

Figure 5.4: **Circle** class that inherits from class **Point**.

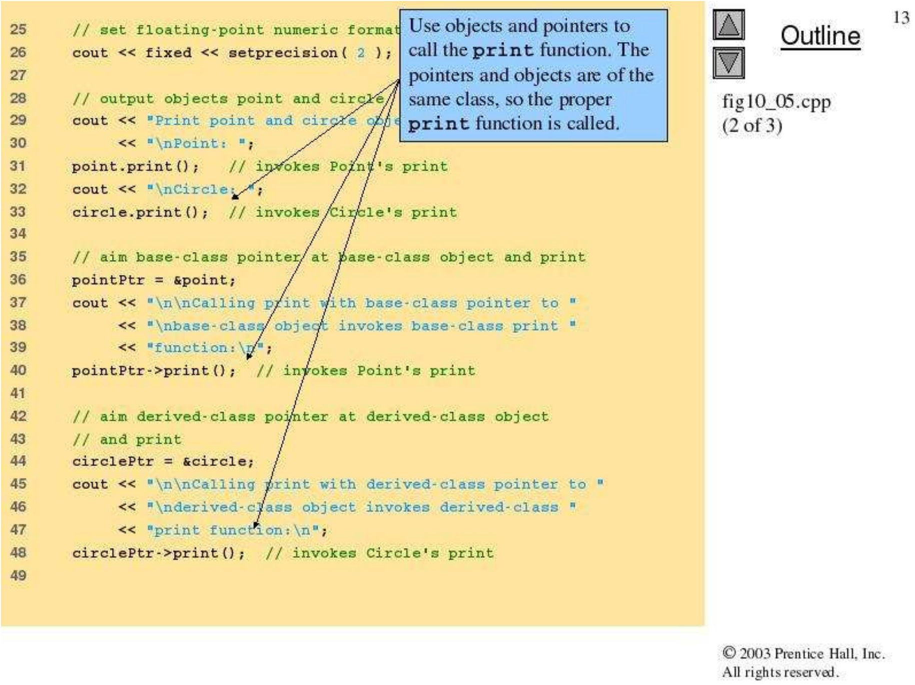


12

Outline

fig10_05.cpp
(1 of 3)

```
// Fig. 10.5: fig10_05.cpp
// Aiming base-class and derived-class pointers at base-class
// and derived-class objects, respectively.
#include <iostream>

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

#include <iomanip>

using std::setprecision;

#include "point.h"   // Point class definition
#include "circle.h"  // Circle class definition

int main()
{
   Point point( 30, 50 );
   Point *pointPtr = 0;    // base-class pointer

   Circle circle( 120, 89, 2.7 );
   Circle *circlePtr = 0;  // derived-class pointer

```



13

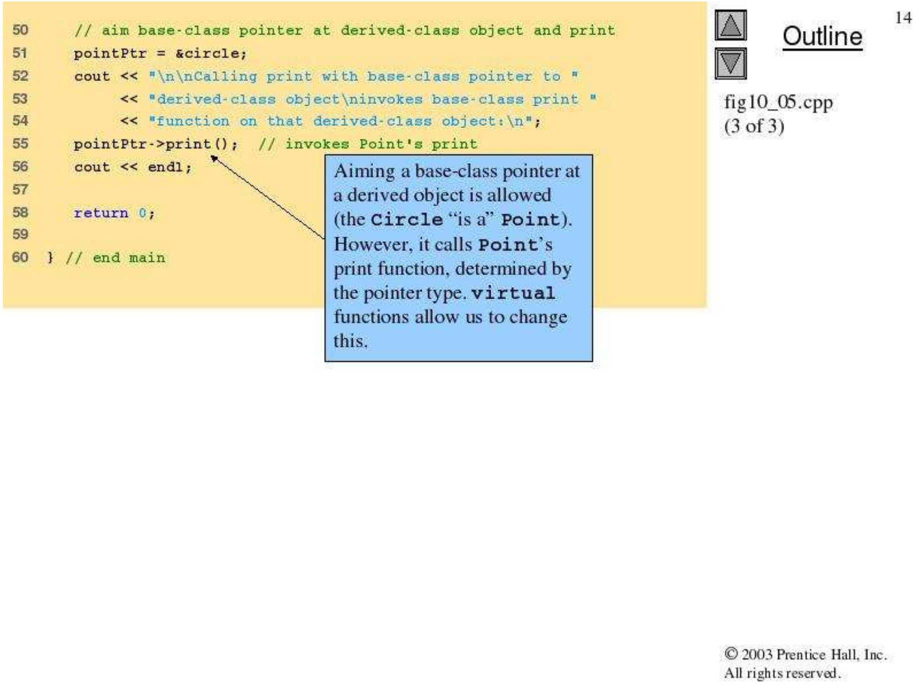
Outline

fig10_05.cpp
(2 of 3)

```
   // set floating-point numeric format
   cout << fixed << setprecision( 2 );

   // output objects point and circle
   cout << "Print point and circle obje
        << "\nPoint: ";
   point.print();   // invokes Point's print
   cout << "\nCircle: ";
   circle.print();  // invokes Circle's print

   // aim base-class pointer at base-class object and print
   pointPtr = &point;
   cout << "\n\nCalling print with base-class pointer to "
        << "\nbase-class object invokes base-class print "
        << "function:\n";
   pointPtr->print();  // invokes Point's print

   // aim derived-class pointer at derived-class object
   // and print
   circlePtr = &circle;
   cout << "\n\nCalling print with derived-class pointer to "
        << "\nderived-class object invokes derived-class "
        << "print function:\n";
   circlePtr->print();  // invokes Circle's print

```

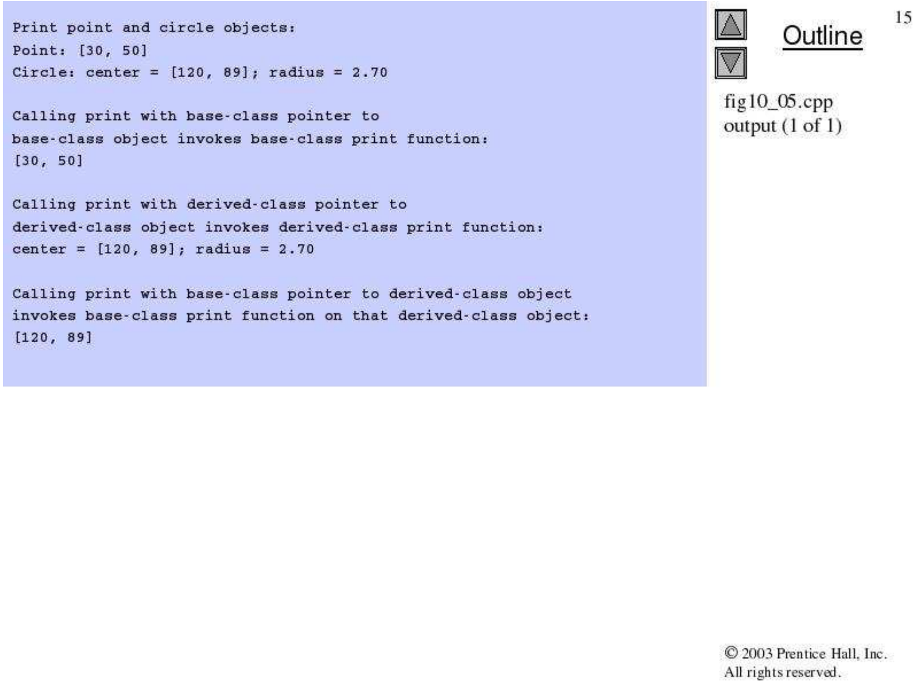
Use objects and pointers to call the **print** function. The pointers and objects are of the same class, so the proper **print** function is called.

Figure 5.5: Assigning addresses of base-class and derived-class objects to base-class and derived-class pointers. (part 1 of 2)



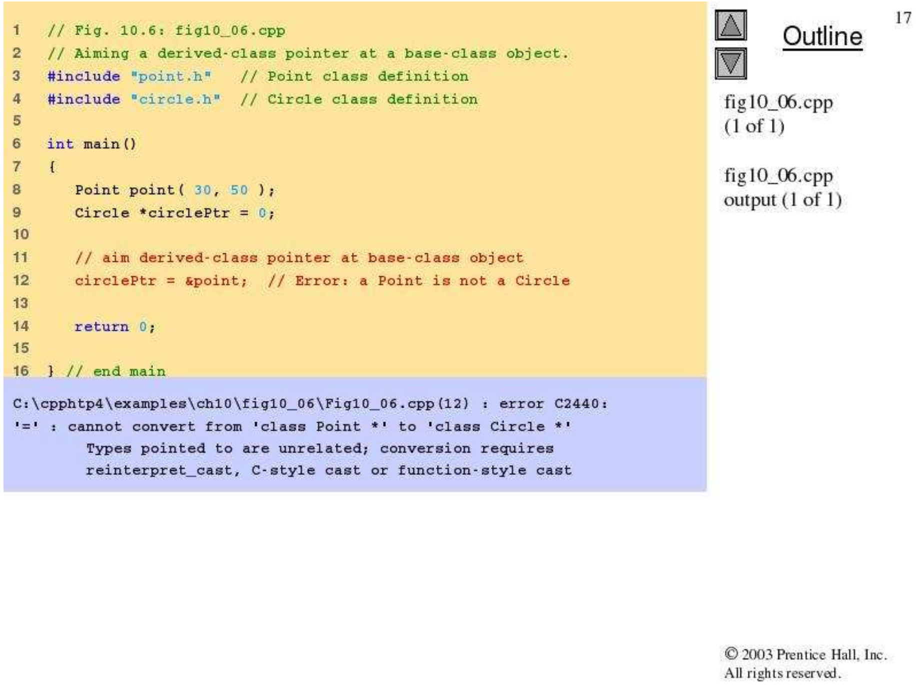
```
   // aim base-class pointer at derived-class object and print
   pointPtr = &circle;
   cout << "\n\nCalling print with base-class pointer to "
        << "derived-class object\ninvokes base-class print "
        << "function on that derived-class object:\n";
   pointPtr->print();  // invokes Point's print
   cout << endl;

   return 0;

} // end main
```

Aiming a base-class pointer at a derived object is allowed (the **Circle** "is a" **Point**). However, it calls **Point**'s print function, determined by the pointer type. **virtual** functions allow us to change this.



```
Print point and circle objects:
Point: [30, 50]
Circle: center = [120, 89]; radius = 2.70

Calling print with base-class pointer to
base-class object invokes base-class print function:
[30, 50]

Calling print with derived-class pointer to
derived-class object invokes derived-class print function:
center = [120, 89]; radius = 2.70

Calling print with base-class pointer to derived-class object
invokes base-class print function on that derived-class object:
[120, 89]
```

Figure 5.6: Assigning addresses of base-class and derived-class objects to base-class and derived-class pointers. (part 2 of 2)



```
// Fig. 10.6: fig10_06.cpp
// Aiming a derived-class pointer at a base-class object.
#include "point.h"   // Point class definition
#include "circle.h"  // Circle class definition

int main()
{
   Point point( 30, 50 );
   Circle *circlePtr = 0;

   // aim derived-class pointer at base-class object
   circlePtr = &point;  // Error: a Point is not a Circle

   return 0;

} // end main
```

```
C:\cpphtp4\examples\ch10\fig10_06\Fig10_06.cpp(12) : error C2440:
'=' : cannot convert from 'class Point *' to 'class Circle *'
       Types pointed to are unrelated; conversion requires
       reinterpret_cast, C-style cast or function-style cast
```

Figure 5.7: Aiming a derived-class pointer at a base-class object.

- Common theme
  - Data type of pointer/reference determines functions it can call

5.2.4 Virtual Functions

- Typically, pointer-class determines functions
- virtual functions; object (not pointer) determines function called
- Why useful?
  - Suppose **Circle**, **Triangle**, **Rectangle** derived from **Shape**; each has own **draw** function
  - To draw any shape

    - Have base class **Shape** pointer, call **draw**
    - Program determines proper **draw** function at run time (dynamically)
    - Treat all shapes generically

- Declare draw as virtual in base class
  - Override **draw** in each derived class; like redefining, but new function must have same signature
  - If function declared **virtual**, can only be overridden
    - **virtual void draw() const;**
    - Once declared **virtual**, **virtual** in all derived classes; good practice to explicitly declare **virtual**
- Dynamic binding
  - Choose proper function to call at run time
  - Only occurs off pointer handles; if function called from object, uses that object"s definition
- Example
  - Redo **Point**, **Circle** example with **virtual** functions
  - Base-class pointer to derived-class object; will call derived-class function
- Polymorphism
  - Same message, "print", given to many objects; all through a base pointer
  - Message takes on "many forms"
- Summary
  - Base-pointer to base-object, derived-pointer to derived; straightforward
  - Base-pointer to derived object; can only call base-class functions
  - Derived-pointer to base-object
    - Compiler error
    - Allowed if explicit cast made (more in section 10.9)

5.3 Polymorphism Examples

- Suppose **Rectangle** derives from **Quadrilateral**
  - **Rectangle** more specific **Quadrilateral**
  - Any operation on **Quadrilateral** can be done on **Rectangle** (i.e., perimeter, area)
- Suppose designing video game
  - Base class **SpaceObject**
    - Derived **Martian**, **SpaceShip**, **LaserBeam**
    - Base function **draw**
  - To refresh screen
    - Screen manager has **vector** of base-class pointers to objects
    - Send **draw** message to each object
    - Same message has "many forms" of results

5.4 Type Fields and switch Structures

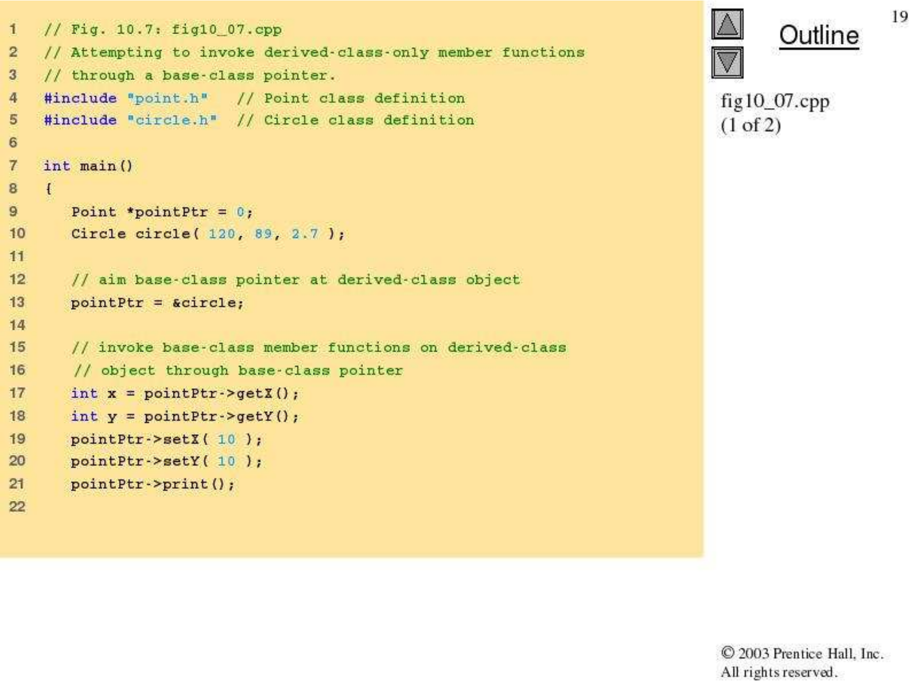
- One way to determine object's class
  - Give base class an attribute; **shapeType** in class **Shape**
  - Use **switch** to call proper **print** function
- Many problems
  - May forget to test for case in **switch**
  - If add/remove a class, must update **switch** structures; Time consuming and error prone
- Better to use polymorphism
  - Less branching logic, simpler programs, less debugging

5.5 Abstract Classes

- Abstract classes
  - Sole purpose: to be a base class (called abstract base classes)

  - Incomplete; derived classes fill in “missing pieces”
  - Cannot make objects from abstract class; however, can have pointers and references

- Concrete classes
  - Can instantiate objects
  - Implement all functions they define
  - Provide specifics
- Abstract classes not required, but helpful
- To make a class abstract
  - Need one or more “pure” virtual functions
    - Declare function with initializer of 0
    - **virtual void draw() const = 0;**
  - Regular virtual functions; have implementations, overriding is optional
  - Pure virtual functions; no implementation, must be overridden
  - Abstract classes can have data and concrete functions; required to have one or more pure virtual functions
- Abstract base class pointers; useful for polymorphism
- Application example
  - Abstract class **Shape**; defines **draw** as pure virtual function
  - **Circle**, **Triangle**, **Rectangle** derived from **Shape**; each must implement **draw**
  - Screen manager knows that each object can draw itself
- Iterators (more Chapter 21)
  - Walk through elements in **vector**/array
  - Use base-class pointer to send **draw** message to each



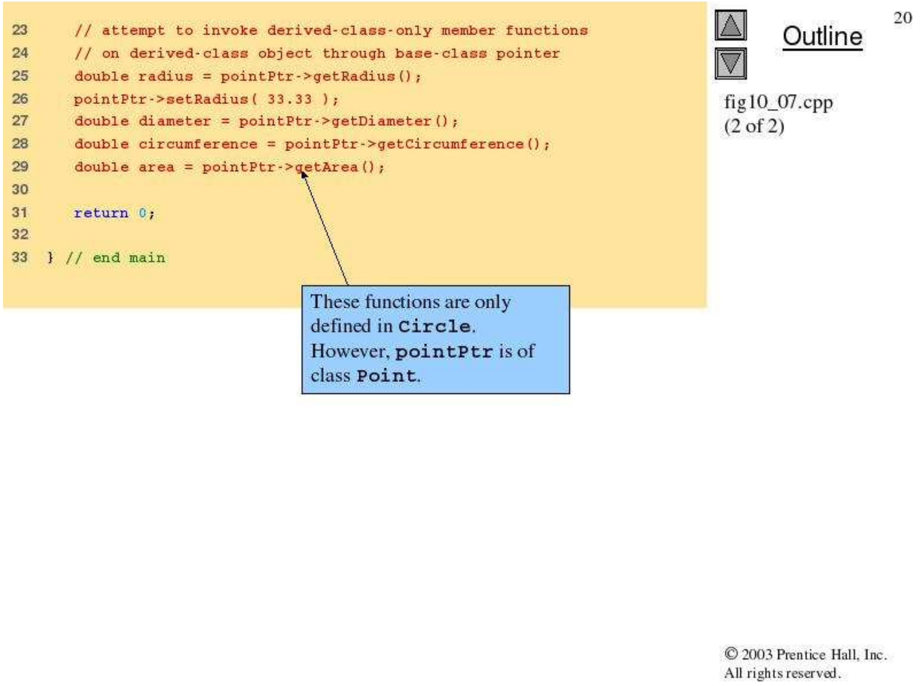
19

Outline

fig10_07.cpp
(1 of 2)

```
// Fig. 10.7: fig10_07.cpp
// Attempting to invoke derived-class-only member functions
// through a base-class pointer.
#include "point.h"   // Point class definition
#include "circle.h"  // Circle class definition

int main()
{
   Point *pointPtr = 0;
   Circle circle( 120, 89, 2.7 );

   // aim base-class pointer at derived-class object
   pointPtr = &circle;

   // invoke base-class member functions on derived-class
   // object through base-class pointer
   int x = pointPtr->getX();
   int y = pointPtr->getY();
   pointPtr->setX( 10 );
   pointPtr->setY( 10 );
   pointPtr->print();

```

Figure 5.8: Attempting to invoke derived-class-only functions via a base-class pointer. (part 1 of 2)

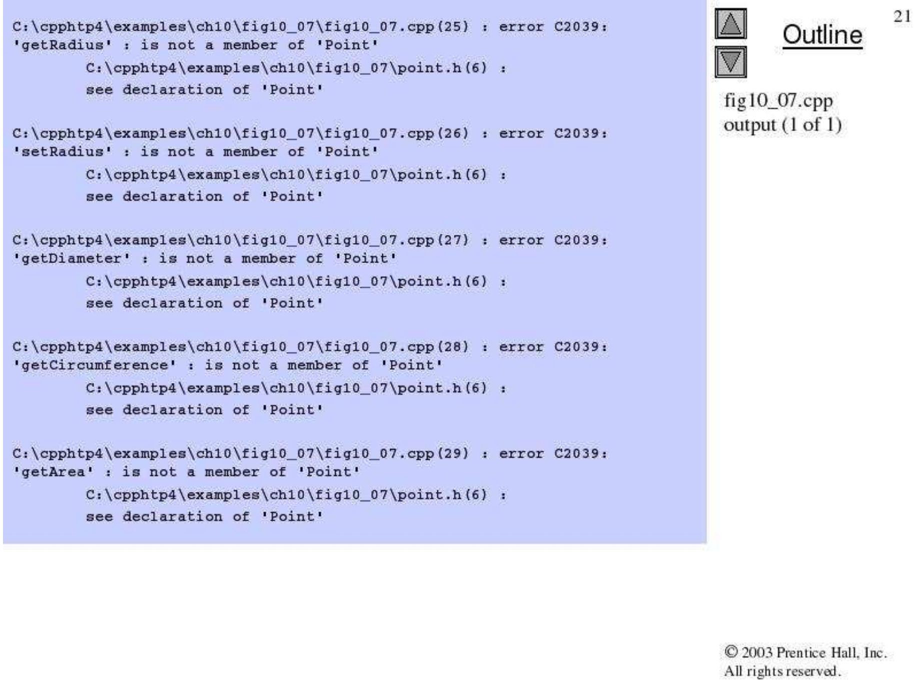


```
23      // attempt to invoke derived-class-only member functions
24      // on derived-class object through base-class pointer
25      double radius = pointPtr->getRadius();
26      pointPtr->setRadius( 33.33 );
27      double diameter = pointPtr->getDiameter();
28      double circumference = pointPtr->getCircumference();
29      double area = pointPtr->getArea();
30
31      return 0;
32
33   } // end main
```

These functions are only defined in **`Circle`**. However, **`pointPtr`** is of class **`Point`**.

fig10_07.cpp
(2 of 2)

20

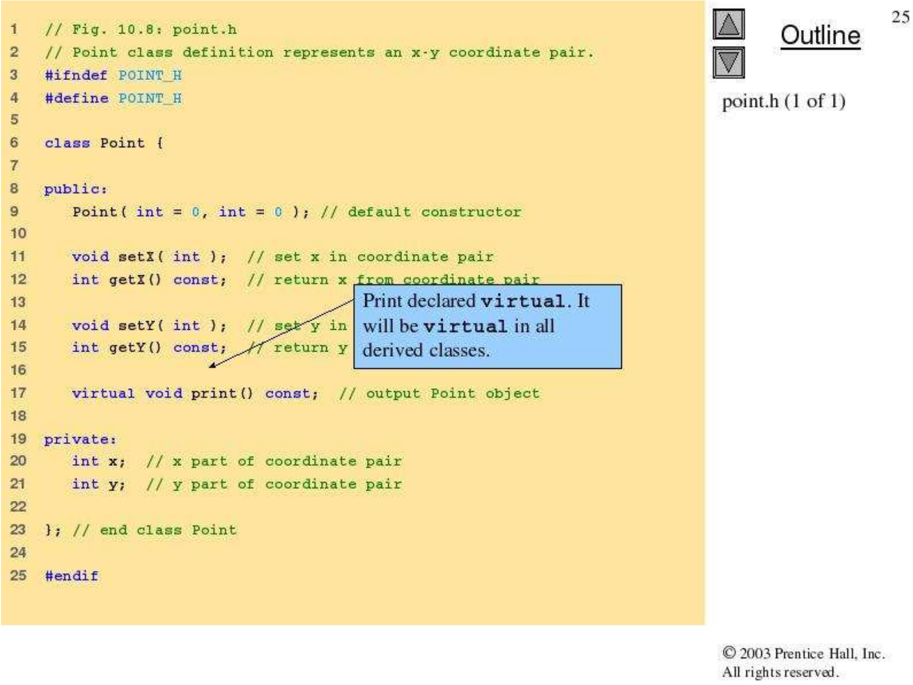


```
C:\cpphtp4\examples\ch10\fig10_07\fig10_07.cpp(25) : error C2039:
'getRadius' : is not a member of 'Point'
        C:\cpphtp4\examples\ch10\fig10_07\point.h(6) :
        see declaration of 'Point'

C:\cpphtp4\examples\ch10\fig10_07\fig10_07.cpp(26) : error C2039:
'setRadius' : is not a member of 'Point'
        C:\cpphtp4\examples\ch10\fig10_07\point.h(6) :
        see declaration of 'Point'

C:\cpphtp4\examples\ch10\fig10_07\fig10_07.cpp(27) : error C2039:
'getDiameter' : is not a member of 'Point'
        C:\cpphtp4\examples\ch10\fig10_07\point.h(6) :
        see declaration of 'Point'

C:\cpphtp4\examples\ch10\fig10_07\fig10_07.cpp(28) : error C2039:
'getCircumference' : is not a member of 'Point'
        C:\cpphtp4\examples\ch10\fig10_07\point.h(6) :
        see declaration of 'Point'

C:\cpphtp4\examples\ch10\fig10_07\fig10_07.cpp(29) : error C2039:
'getArea' : is not a member of 'Point'
        C:\cpphtp4\examples\ch10\fig10_07\point.h(6) :
        see declaration of 'Point'
```

fig10_07.cpp
output (1 of 1)

21

Figure 5.9: Attempting to invoke derived-class-only functions via a base-class pointer. (part 2 of 2)



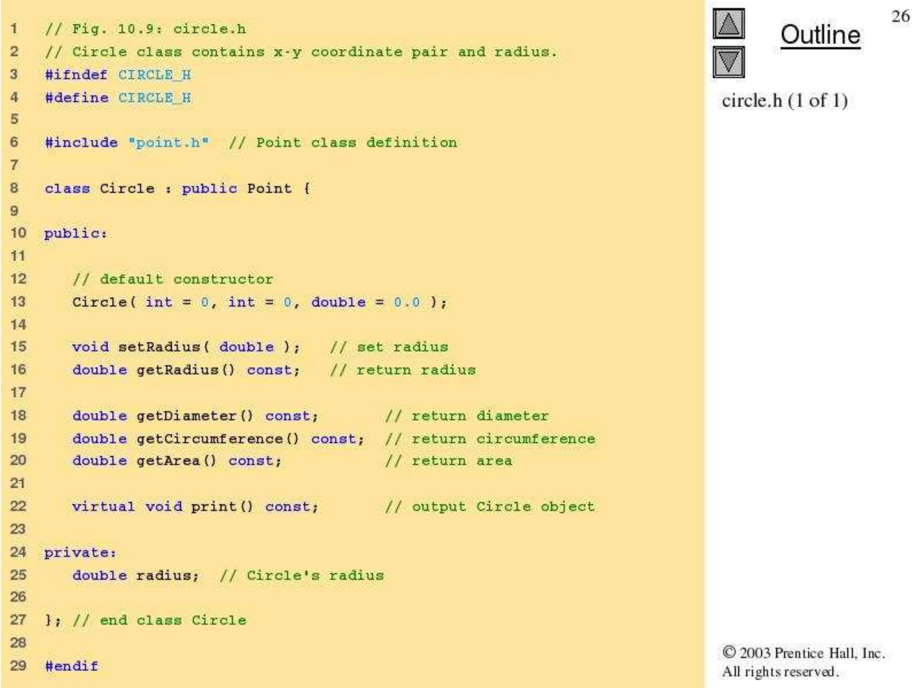
```
// Fig. 10.8: point.h
// Point class definition represents an x-y coordinate pair.
#ifndef POINT_H
#define POINT_H

class Point {

public:
   Point( int = 0, int = 0 ); // default constructor

   void setX( int );  // set x in coordinate pair
   int getX() const;  // return x from coordinate pair

   void setY( int );  // set y in coordinate pair
   int getY() const;  // return y from coordinate pair

   virtual void print() const;  // output Point object

private:
   int x;  // x part of coordinate pair
   int y;  // y part of coordinate pair

}; // end class Point

#endif
```

Print declared **virtual**. It will be **virtual** in all derived classes.

25

Outline

point.h (1 of 1)



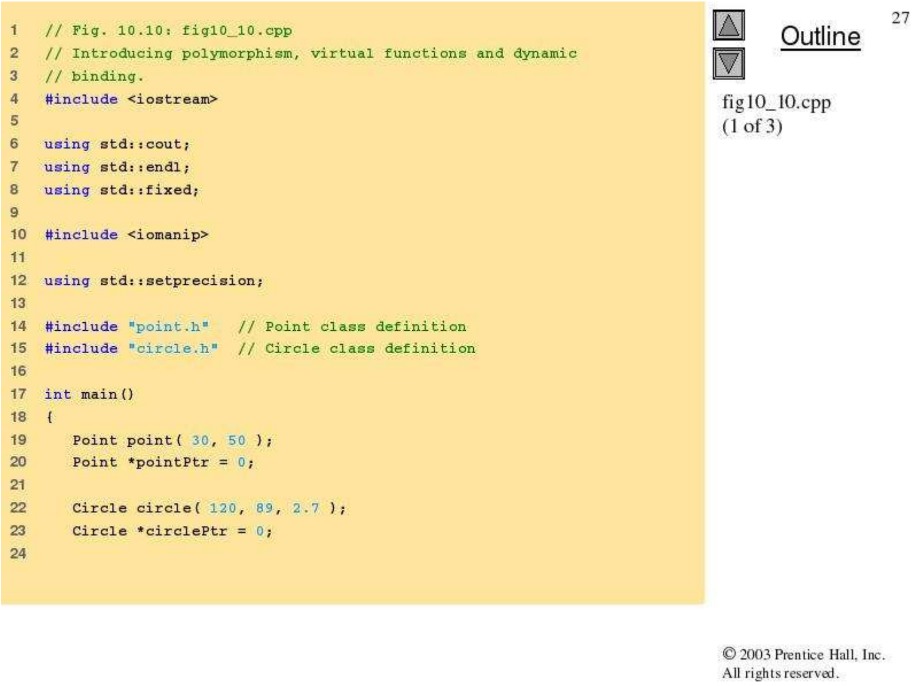
```
// Fig. 10.9: circle.h
// Circle class contains x-y coordinate pair and radius.
#ifndef CIRCLE_H
#define CIRCLE_H

#include "point.h"  // Point class definition

class Circle : public Point {

public:

   // default constructor
   Circle( int = 0, int = 0, double = 0.0 );

   void setRadius( double );   // set radius
   double getRadius() const;   // return radius

   double getDiameter() const;       // return diameter
   double getCircumference() const;  // return circumference
   double getArea() const;           // return area

   virtual void print() const;       // output Circle object

private:
   double radius;  // Circle's radius

}; // end class Circle

#endif
```

26

Outline

circle.h (1 of 1)

Figure 5.10: **Point** class header file declares **print** function as **virtual** (upper) and **Circle** class header file declares **print** function as **virtual**.

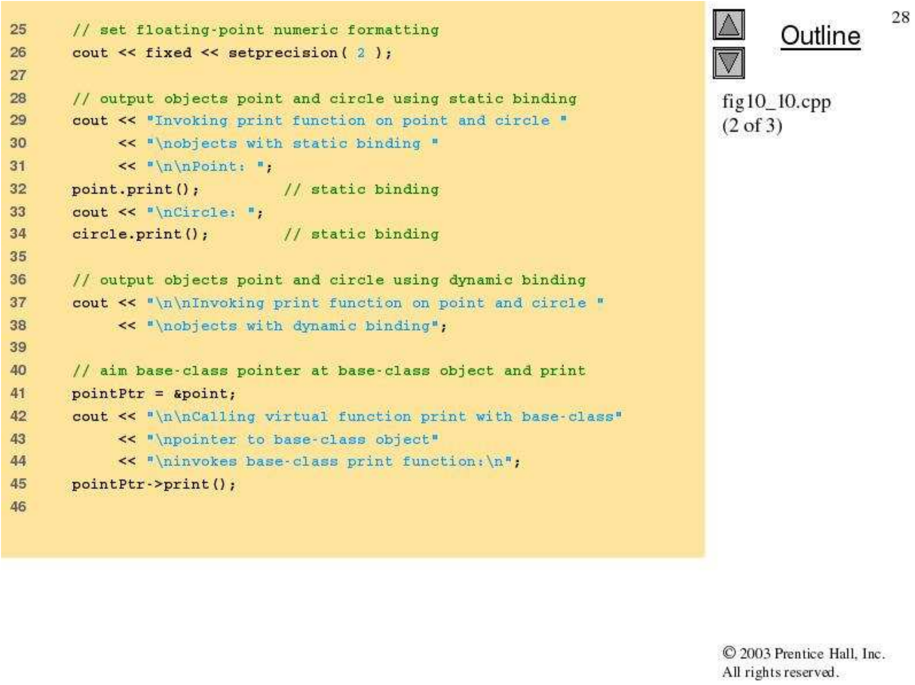


```
// Fig. 10.10: fig10_10.cpp
// Introducing polymorphism, virtual functions and dynamic
// binding.
#include <iostream>

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

#include <iomanip>

using std::setprecision;

#include "point.h"   // Point class definition
#include "circle.h"  // Circle class definition

int main()
{
   Point point( 30, 50 );
   Point *pointPtr = 0;

   Circle circle( 120, 89, 2.7 );
   Circle *circlePtr = 0;

```

27

Outline

fig10_10.cpp
(1 of 3)



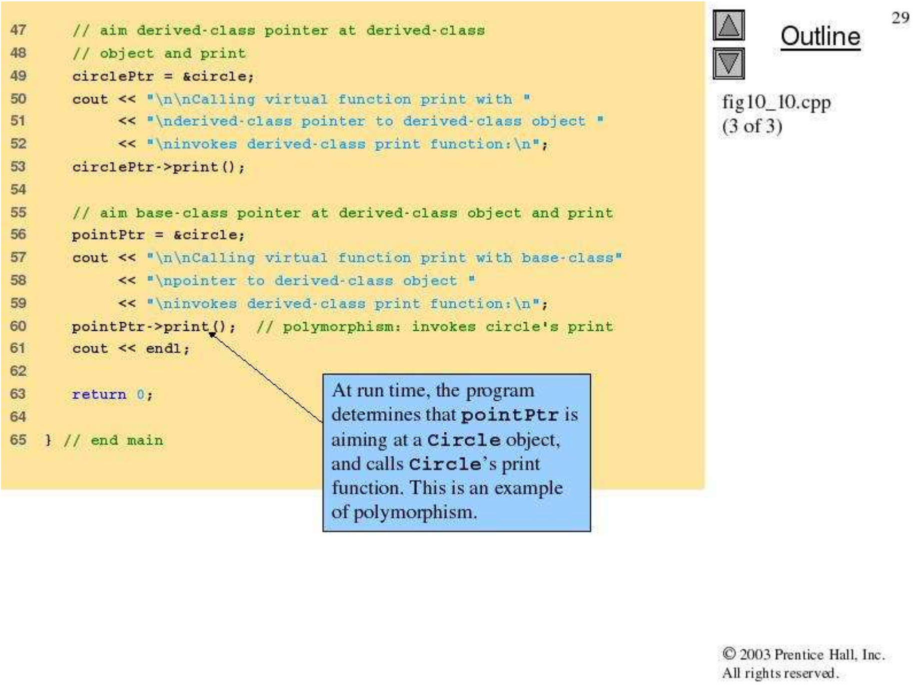
```
   // set floating-point numeric formatting
   cout << fixed << setprecision( 2 );

   // output objects point and circle using static binding
   cout << "Invoking print function on point and circle "
        << "\nobjects with static binding "
        << "\n\nPoint: ";
   point.print();         // static binding
   cout << "\nCircle: ";
   circle.print();        // static binding

   // output objects point and circle using dynamic binding
   cout << "\n\nInvoking print function on point and circle "
        << "\nobjects with dynamic binding";

   // aim base-class pointer at base-class object and print
   pointPtr = &point;
   cout << "\n\nCalling virtual function print with base-class"
        << "\npointer to base-class object"
        << "\ninvokes base-class print function:\n";
   pointPtr->print();

```

28

Outline

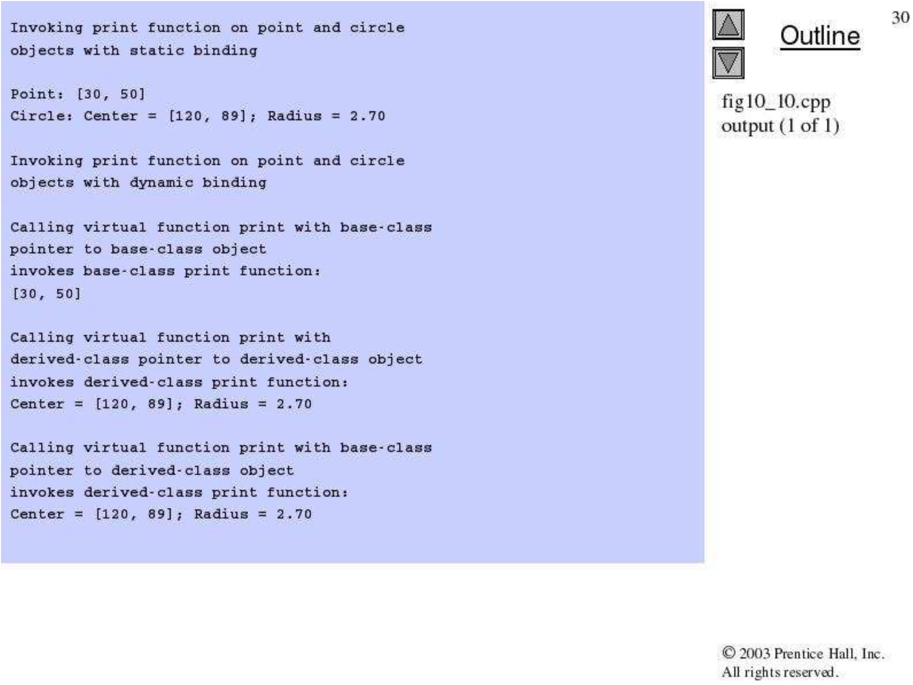
fig10_10.cpp
(2 of 3)

Figure 5.11: Demonstrating polymorphism by invoking a derived-class virtual function via a base-class pointer to a derived-class object. (part 1 of 2)



```
   // aim derived-class pointer at derived-class
   // object and print
   circlePtr = &circle;
   cout << "\n\nCalling virtual function print with "
        << "\nderived-class pointer to derived-class object "
        << "\ninvokes derived-class print function:\n";
   circlePtr->print();

   // aim base-class pointer at derived-class object and print
   pointPtr = &circle;
   cout << "\n\nCalling virtual function print with base-class"
        << "\npointer to derived-class object "
        << "\ninvokes derived-class print function:\n";
   pointPtr->print();  // polymorphism: invokes circle's print
   cout << endl;

   return 0;

} // end main
```

At run time, the program determines that **pointPtr** is aiming at a **Circle** object, and calls **Circle**'s print function. This is an example of polymorphism.



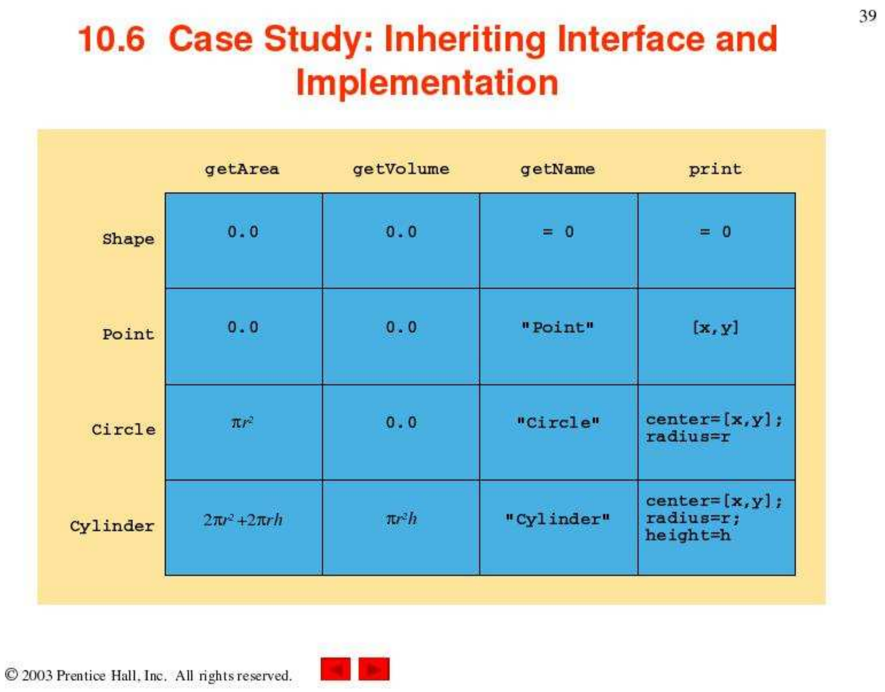
```
Invoking print function on point and circle
objects with static binding

Point: [30, 50]
Circle: Center = [120, 89]; Radius = 2.70

Invoking print function on point and circle
objects with dynamic binding

Calling virtual function print with base-class
pointer to base-class object
invokes base-class print function:
[30, 50]

Calling virtual function print with
derived-class pointer to derived-class object
invokes derived-class print function:
Center = [120, 89]; Radius = 2.70

Calling virtual function print with base-class
pointer to derived-class object
invokes derived-class print function:
Center = [120, 89]; Radius = 2.70
```

Figure 5.12: Demonstrating polymorphism by invoking a derived-class virtual function via a base-class pointer to a derived-class object. (part 2 of 2)

5.6 Case Study: Inheriting Interface and Implementation

Make abstract base class Shape

- Pure virtual functions (must be implemented)
  - **getName**, **print**
  - Default implementation does not make sense
- Virtual functions (may be redefined)
  - **getArea**, **getVolume**; initially return **0.0**
  - If not redefined, uses base class definition
- Derive classes **Point**, **Circle**, **Cylinder**



39

10.6 Case Study: Inheriting Interface and Implementation

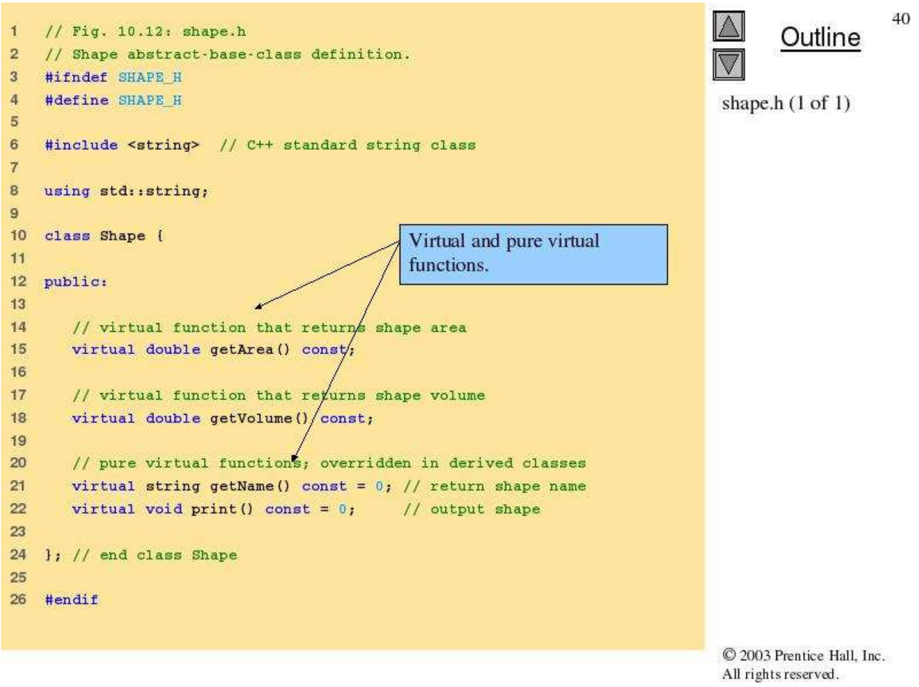
	getArea	getVolume	getName	print
Shape	0.0	0.0	= 0	= 0
Point	0.0	0.0	"Point"	[x,y]

Circle	πr^2	0.0	"Circle"	center=[x,y];
				radius=r

Cylinder	$2\pi r^2 + 2\pi rh$	$\pi r^2 h$	"Cylinder"	center=[x,y];
				radius=r;
				height=h

© 2003 Prentice Hall, Inc. All rights reserved.

Figure 5.13: Defining the polymorphic interface for the **Shape** hierarchy classes.



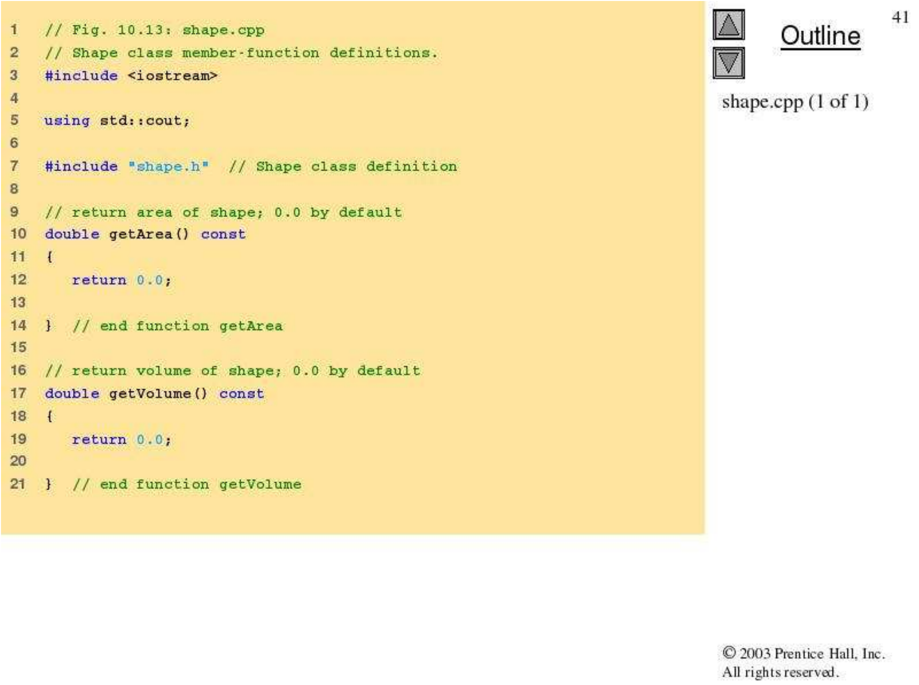
```
// Fig. 10.12: shape.h
// Shape abstract-base-class definition.
#ifndef SHAPE_H
#define SHAPE_H

#include <string>  // C++ standard string class

using std::string;

class Shape {

public:

   // virtual function that returns shape area
   virtual double getArea() const;

   // virtual function that returns shape volume
   virtual double getVolume() const;

   // pure virtual functions; overridden in derived classes
   virtual string getName() const = 0; // return shape name
   virtual void print() const = 0;     // output shape

}; // end class Shape

#endif
```

Virtual and pure virtual functions.

40

Outline

shape.h (1 of 1)



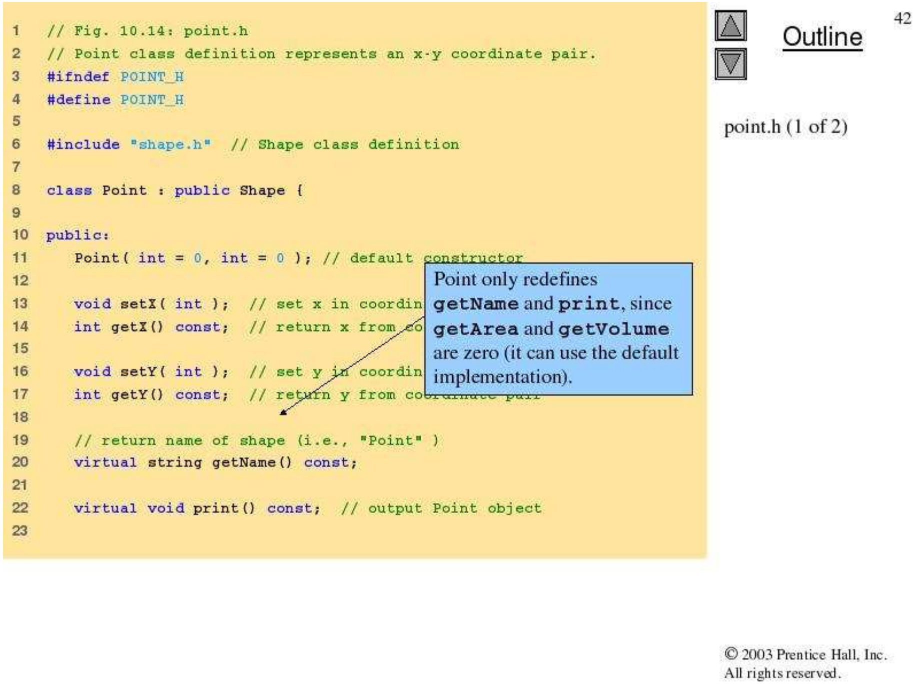
```
// Fig. 10.13: shape.cpp
// Shape class member-function definitions.
#include <iostream>

using std::cout;

#include "shape.h"  // Shape class definition

// return area of shape; 0.0 by default
double getArea() const
{
   return 0.0;

}  // end function getArea

// return volume of shape; 0.0 by default
double getVolume() const
{
   return 0.0;

}  // end function getVolume
```

41

Outline

shape.cpp (1 of 1)

Figure 5.14: Abstract base class **Shape** header file and Abstract base class **Shape**.



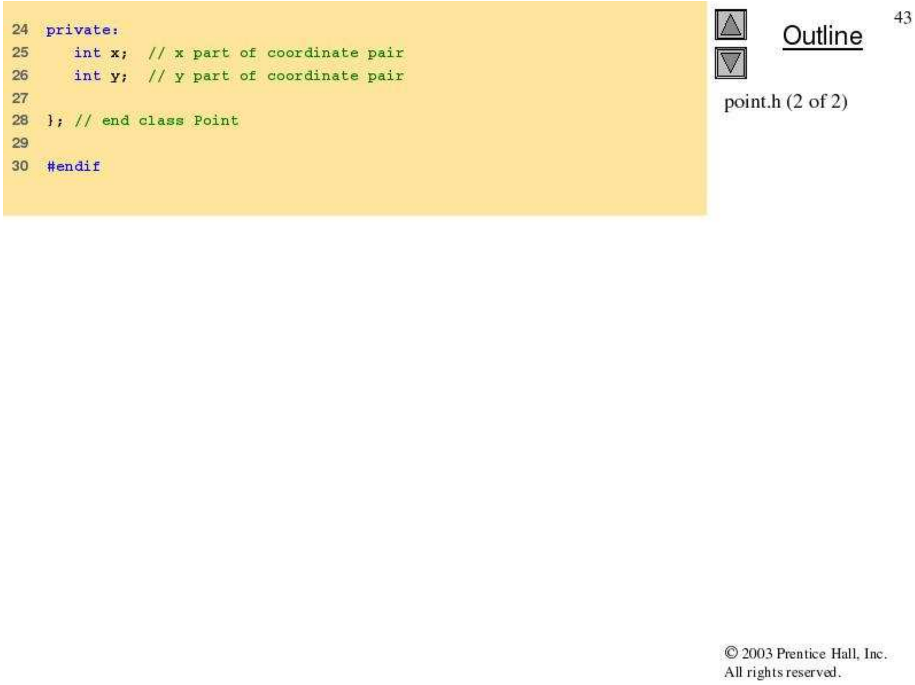
42

Outline

point.h (1 of 2)

```
// Fig. 10.14: point.h
// Point class definition represents an x-y coordinate pair.
#ifndef POINT_H
#define POINT_H

#include "shape.h"  // Shape class definition

class Point : public Shape {

public:
   Point( int = 0, int = 0 ); // default constructor

   void setX( int );  // set x in coordin
   int getX() const;  // return x from co

   void setY( int );  // set y in coordin
   int getY() const;  // return y from coordinate pair

   // return name of shape (i.e., "Point" )
   virtual string getName() const;

   virtual void print() const;  // output Point object

```

Point only redefines **getName** and **print**, since **getArea** and **getVolume** are zero (it can use the default implementation).



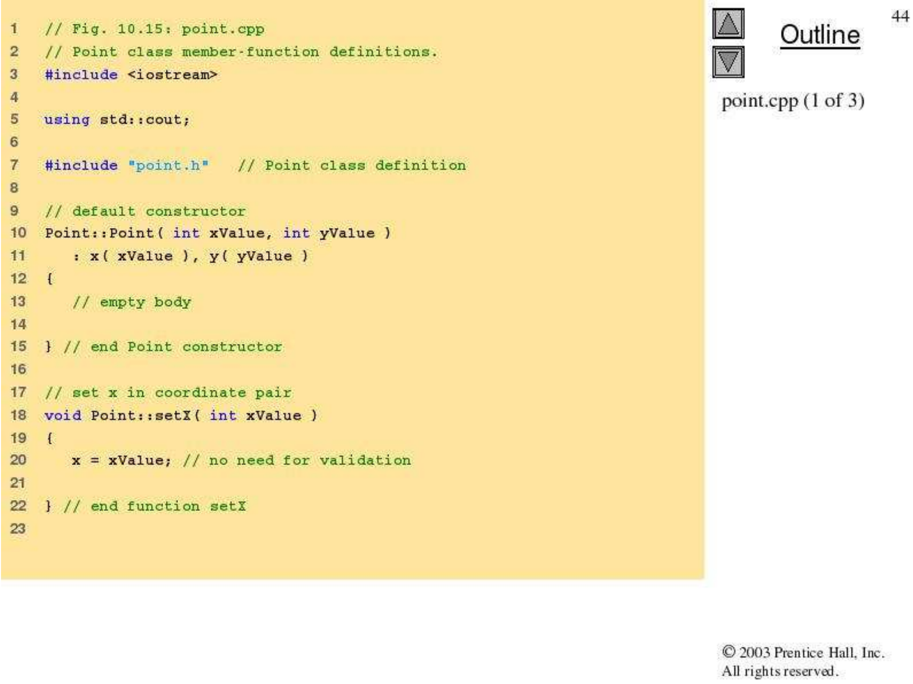
43

Outline

point.h (2 of 2)

```
private:
   int x;  // x part of coordinate pair
   int y;  // y part of coordinate pair

}; // end class Point

#endif
```

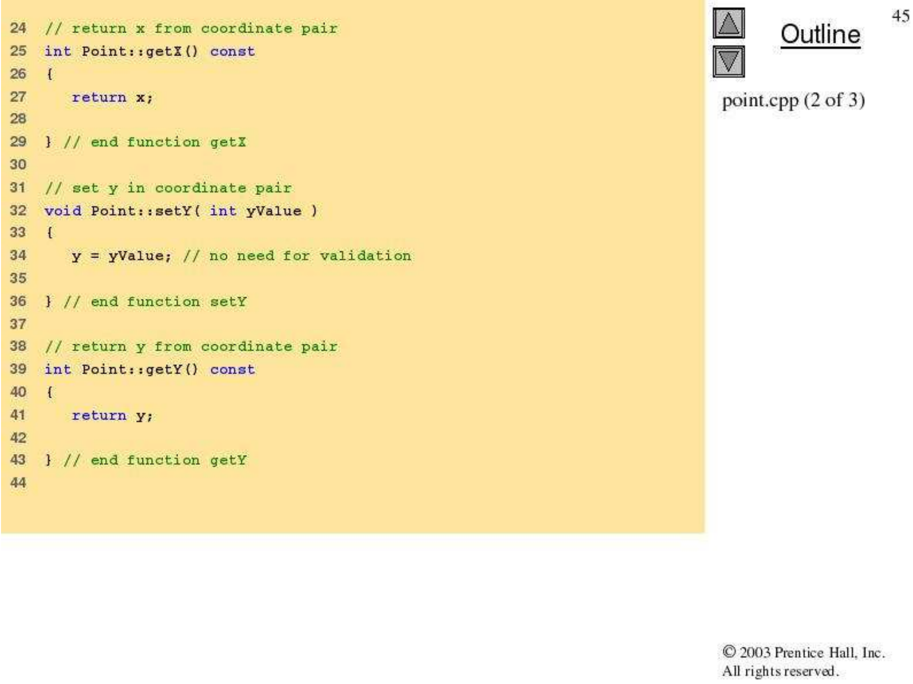
Figure 5.15: **Point** class header file.



point.cpp (1 of 3)

```cpp
// Fig. 10.15: point.cpp
// Point class member-function definitions.
#include <iostream>

using std::cout;

#include "point.h"   // Point class definition

// default constructor
Point::Point( int xValue, int yValue )
   : x( xValue ), y( yValue )
{
   // empty body

} // end Point constructor

// set x in coordinate pair
void Point::setX( int xValue )
{
   x = xValue; // no need for validation

} // end function setX

```

© 2003 Prentice Hall, Inc.
All rights reserved.

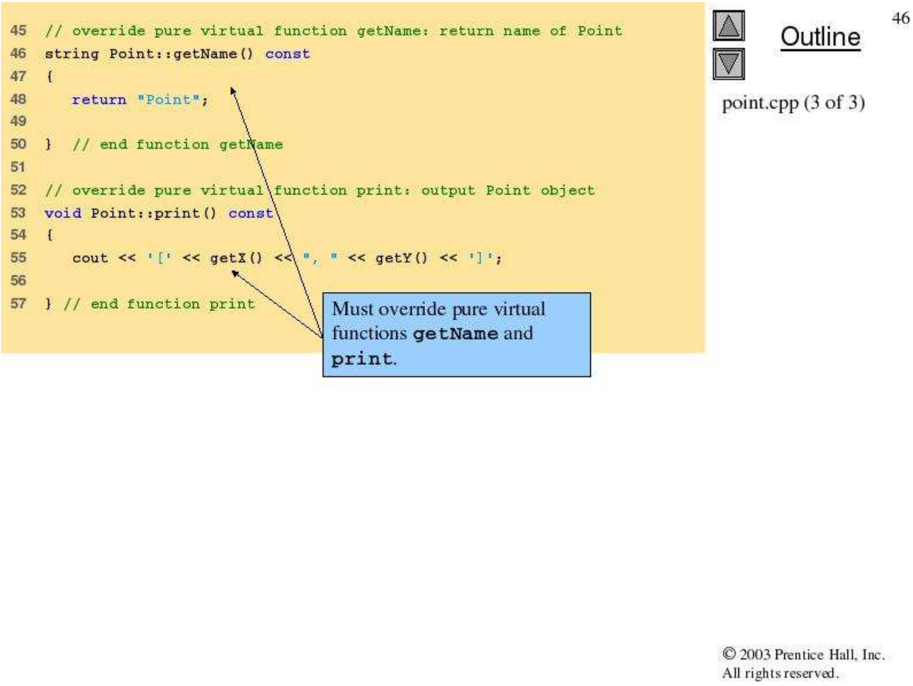


point.cpp (2 of 3)

```cpp
// return x from coordinate pair
int Point::getX() const
{
   return x;

} // end function getX

// set y in coordinate pair
void Point::setY( int yValue )
{
   y = yValue; // no need for validation

} // end function setY

// return y from coordinate pair
int Point::getY() const
{
   return y;

} // end function getY

```

© 2003 Prentice Hall, Inc.
All rights reserved.

Figure 5.16: **Point** class implementation file. (part 1 of 2)



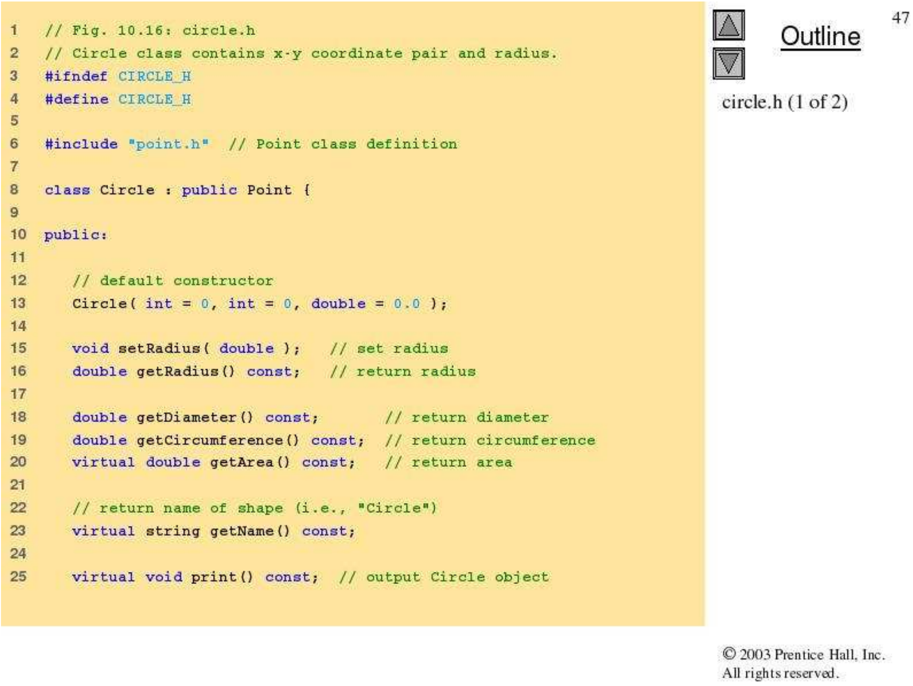
```
// override pure virtual function getName: return name of Point
string Point::getName() const
{
   return "Point";

}  // end function getName

// override pure virtual function print: output Point object
void Point::print() const
{
   cout << '[' << getX() << ", " << getY() << ']';

} // end function print
```

Must override pure virtual functions **getName** and **print**.

46

Outline

point.cpp (3 of 3)



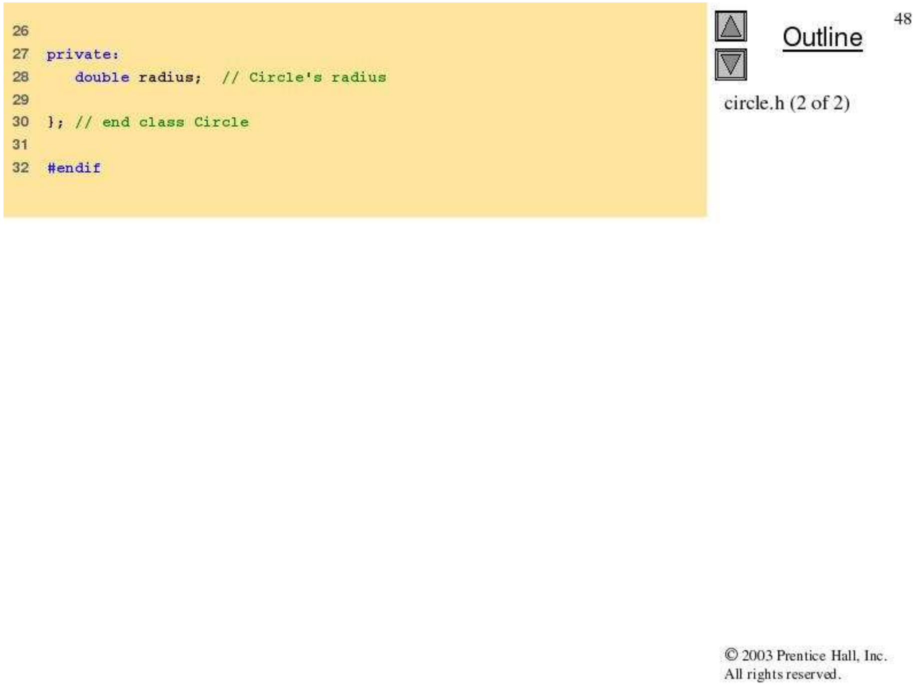
```
// Fig. 10.16: circle.h
// Circle class contains x-y coordinate pair and radius.
#ifndef CIRCLE_H
#define CIRCLE_H

#include "point.h"  // Point class definition

class Circle : public Point {

public:

   // default constructor
   Circle( int = 0, int = 0, double = 0.0 );

   void setRadius( double );   // set radius
   double getRadius() const;   // return radius

   double getDiameter() const;       // return diameter
   double getCircumference() const;  // return circumference
   virtual double getArea() const;   // return area

   // return name of shape (i.e., "Circle")
   virtual string getName() const;

   virtual void print() const;  // output Circle object
```

47

Outline

circle.h (1 of 2)

Figure 5.17: **Point** class implementation file. (part 2 of 2)

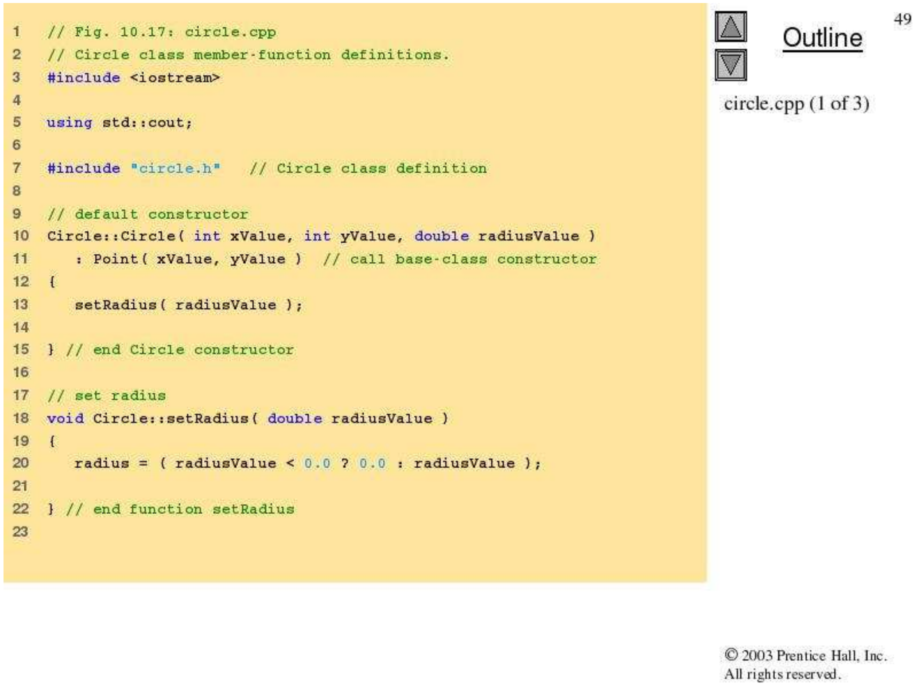


48

Outline

circle.h (2 of 2)

```

private:
   double radius;  // Circle's radius

}; // end class Circle

#endif
```



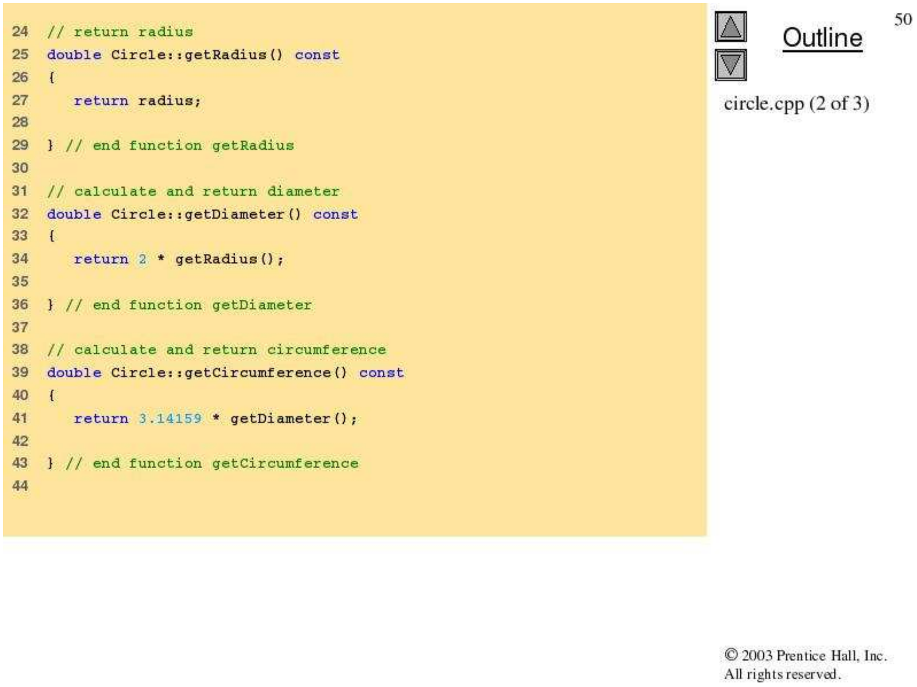
49

Outline

circle.cpp (1 of 3)

```
// Fig. 10.17: circle.cpp
// Circle class member-function definitions.
#include <iostream>

using std::cout;

#include "circle.h"   // Circle class definition

// default constructor
Circle::Circle( int xValue, int yValue, double radiusValue )
   : Point( xValue, yValue )  // call base-class constructor
{
   setRadius( radiusValue );

} // end Circle constructor

// set radius
void Circle::setRadius( double radiusValue )
{
   radius = ( radiusValue < 0.0 ? 0.0 : radiusValue );

} // end function setRadius

```

Figure 5.18: **Circle** class header file and **Circle** class that inherits from class **Point**. (part 1 of 2)

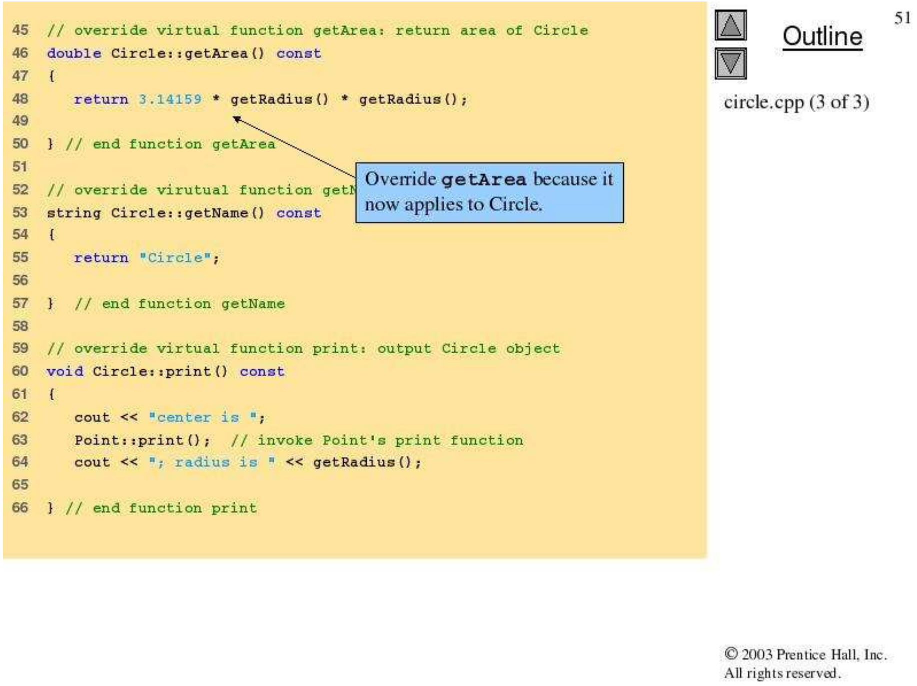


50

Outline

circle.cpp (2 of 3)

```cpp
// return radius
double Circle::getRadius() const
{
   return radius;

} // end function getRadius

// calculate and return diameter
double Circle::getDiameter() const
{
   return 2 * getRadius();

} // end function getDiameter

// calculate and return circumference
double Circle::getCircumference() const
{
   return 3.14159 * getDiameter();

} // end function getCircumference

```



51

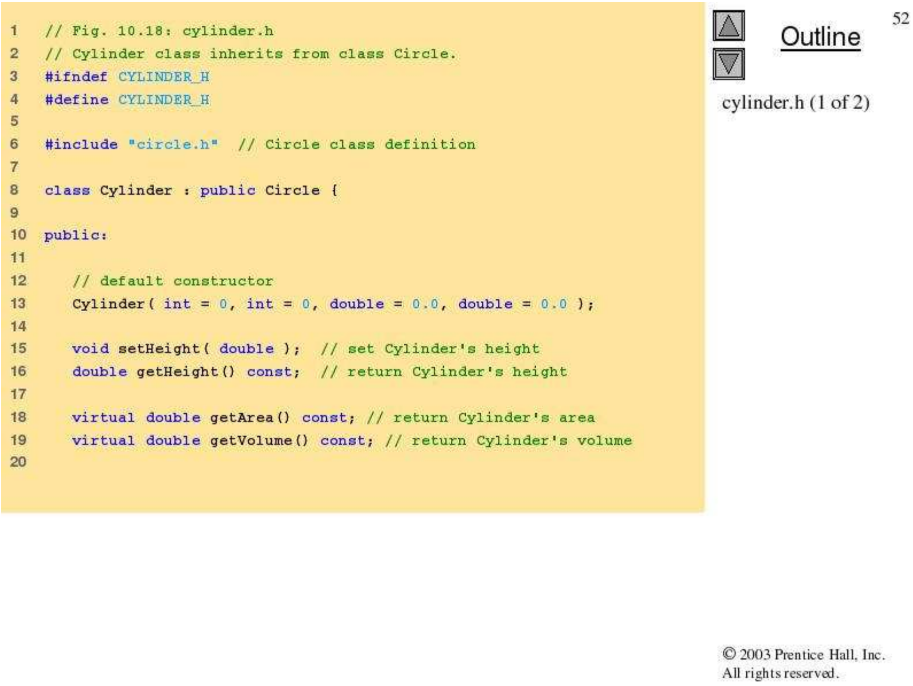
Outline

circle.cpp (3 of 3)

```cpp
// override virtual function getArea: return area of Circle
double Circle::getArea() const
{
   return 3.14159 * getRadius() * getRadius();

} // end function getArea

// override virtual function getN
string Circle::getName() const
{
   return "Circle";

}  // end function getName

// override virtual function print: output Circle object
void Circle::print() const
{
   cout << "center is ";
   Point::print();  // invoke Point's print function
   cout << "; radius is " << getRadius();

} // end function print
```

Override **getArea** because it now applies to Circle.

Figure 5.19: **Circle** class that inherits from class **Point**. (part 2 of 2)

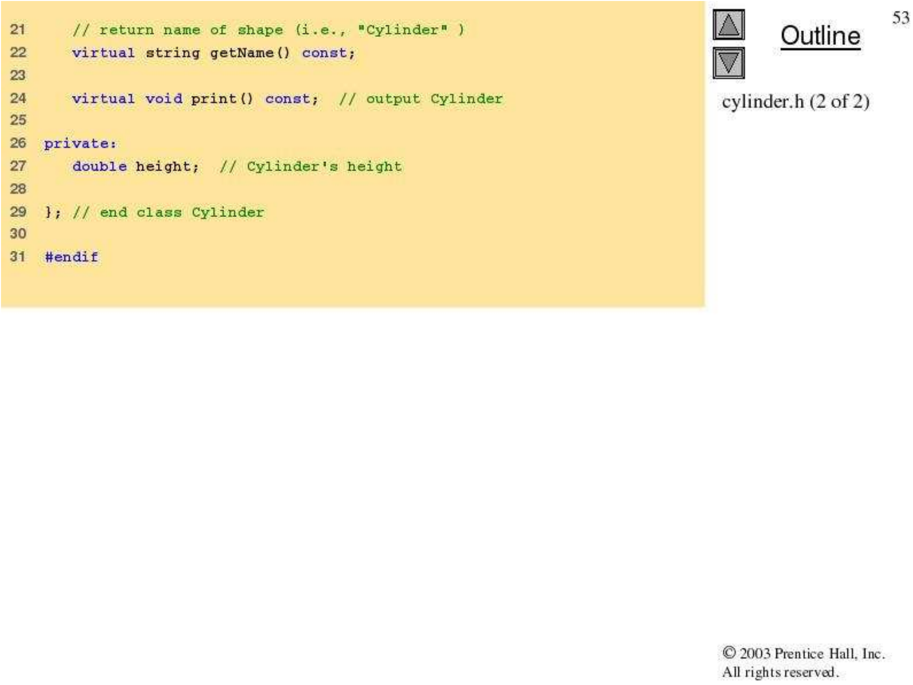


52

Outline

cylinder.h (1 of 2)

```
// Fig. 10.18: cylinder.h
// Cylinder class inherits from class Circle.
#ifndef CYLINDER_H
#define CYLINDER_H

#include "circle.h"  // Circle class definition

class Cylinder : public Circle {

public:

   // default constructor
   Cylinder( int = 0, int = 0, double = 0.0, double = 0.0 );

   void setHeight( double );  // set Cylinder's height
   double getHeight() const;  // return Cylinder's height

   virtual double getArea() const; // return Cylinder's area
   virtual double getVolume() const; // return Cylinder's volume

```



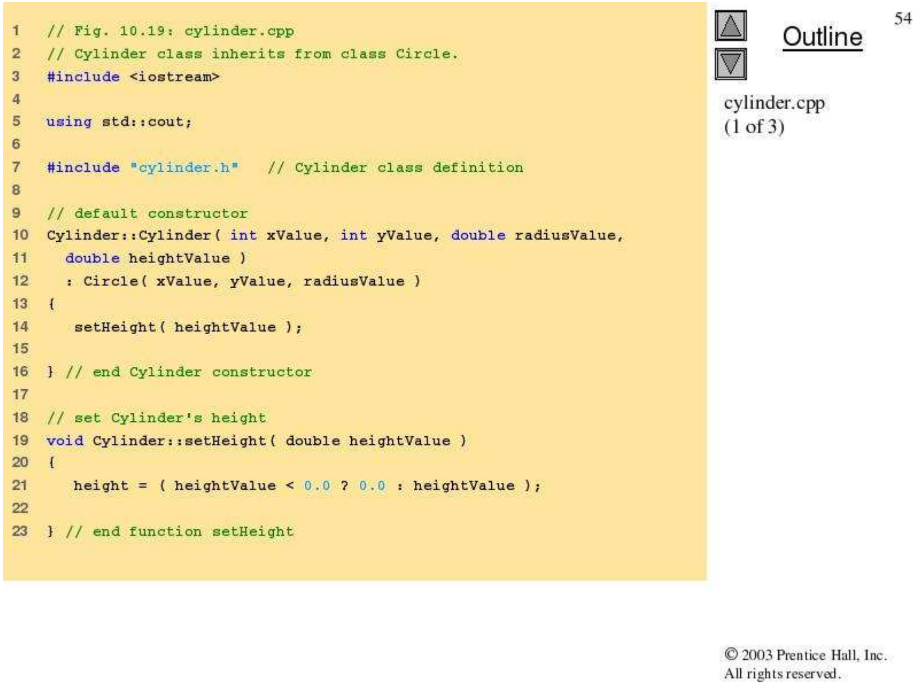
53

Outline

cylinder.h (2 of 2)

```
   // return name of shape (i.e., "Cylinder" )
   virtual string getName() const;

   virtual void print() const;  // output Cylinder

private:
   double height;  // Cylinder's height

}; // end class Cylinder

#endif
```

Figure 5.20: **Cylinder** class header file.

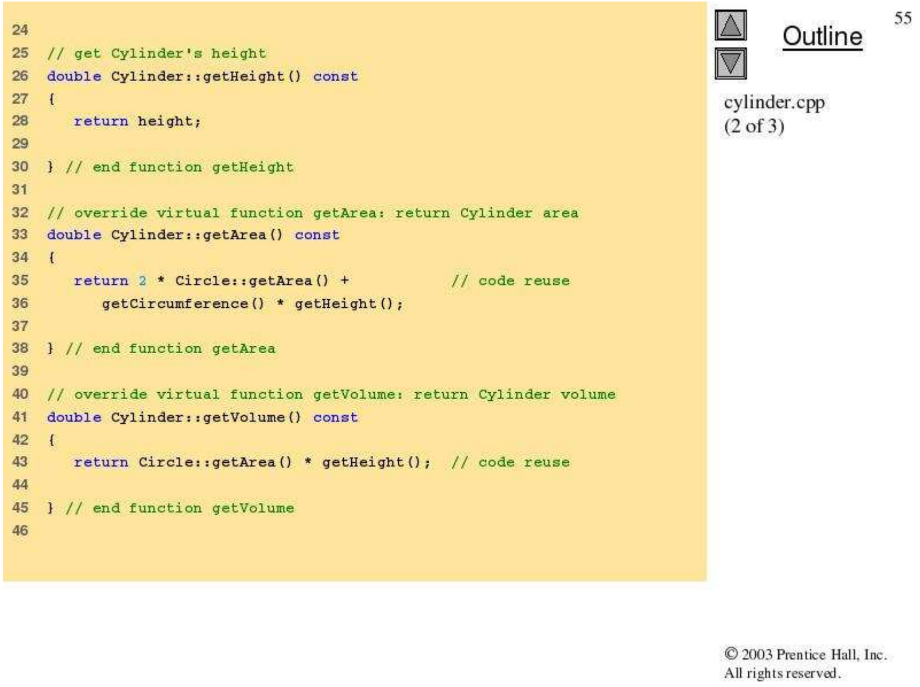


```
// Fig. 10.19: cylinder.cpp
// Cylinder class inherits from class Circle.
#include <iostream>

using std::cout;

#include "cylinder.h"   // Cylinder class definition

// default constructor
Cylinder::Cylinder( int xValue, int yValue, double radiusValue,
  double heightValue )
  : Circle( xValue, yValue, radiusValue )
{
   setHeight( heightValue );

} // end Cylinder constructor

// set Cylinder's height
void Cylinder::setHeight( double heightValue )
{
   height = ( heightValue < 0.0 ? 0.0 : heightValue );

} // end function setHeight
```

54

Outline

cylinder.cpp
(1 of 3)



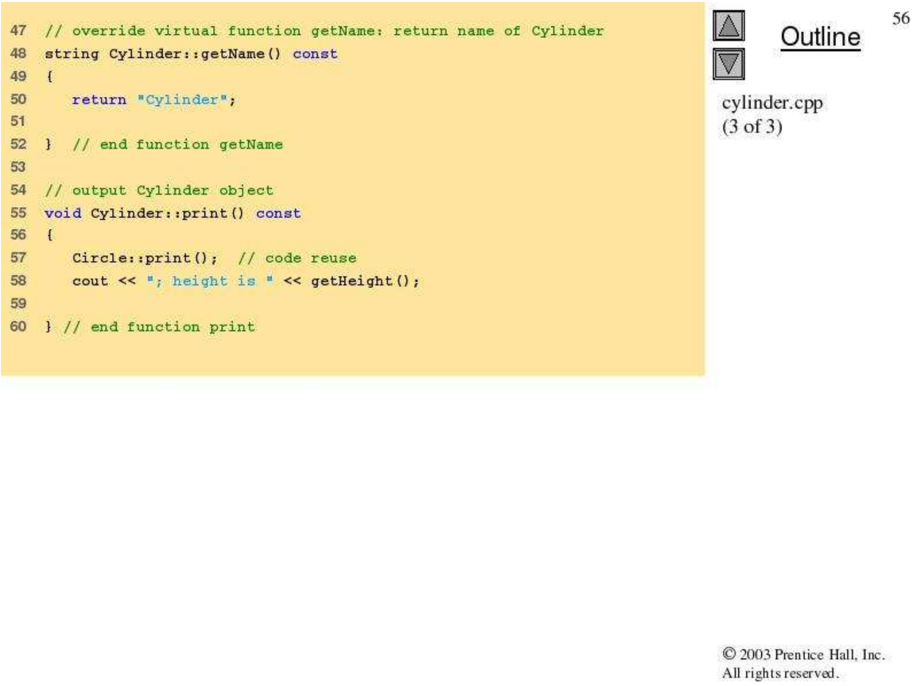
```

// get Cylinder's height
double Cylinder::getHeight() const
{
   return height;

} // end function getHeight

// override virtual function getArea: return Cylinder area
double Cylinder::getArea() const
{
   return 2 * Circle::getArea() +           // code reuse
      getCircumference() * getHeight();

} // end function getArea

// override virtual function getVolume: return Cylinder volume
double Cylinder::getVolume() const
{
   return Circle::getArea() * getHeight();  // code reuse

} // end function getVolume

```

55

Outline

cylinder.cpp
(2 of 3)

Figure 5.21: **Cylinder** class implementation file. (part 1 of 2)

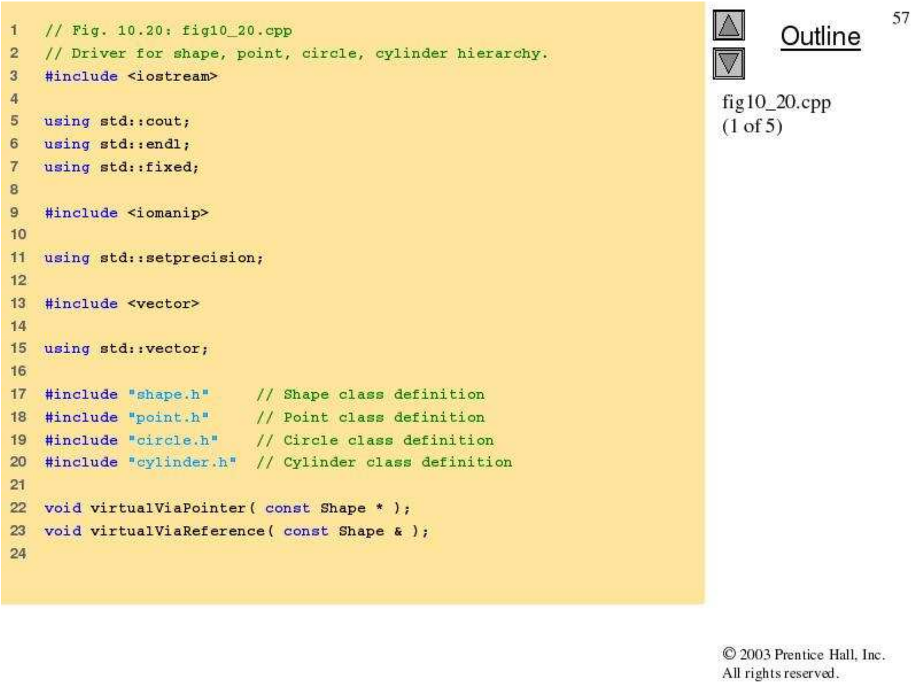


56

Outline

```
// override virtual function getName: return name of Cylinder
string Cylinder::getName() const
{
   return "Cylinder";

}  // end function getName

// output Cylinder object
void Cylinder::print() const
{
   Circle::print();  // code reuse
   cout << "; height is " << getHeight();

} // end function print
```

cylinder.cpp
(3 of 3)



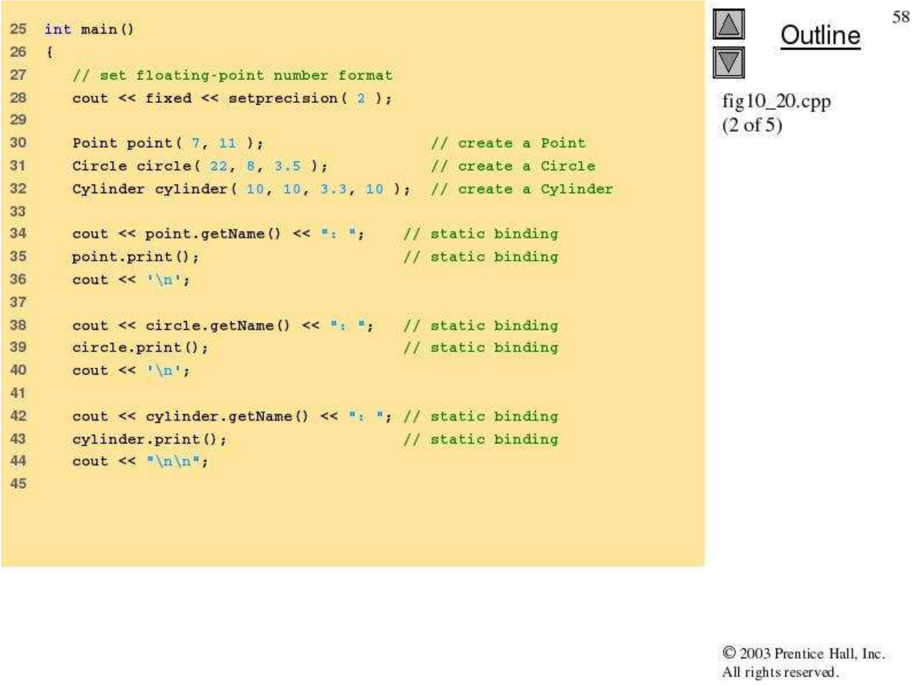
57

Outline

```
// Fig. 10.20: fig10_20.cpp
// Driver for shape, point, circle, cylinder hierarchy.
#include <iostream>

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

#include <iomanip>

using std::setprecision;

#include <vector>

using std::vector;

#include "shape.h"     // Shape class definition
#include "point.h"     // Point class definition
#include "circle.h"    // Circle class definition
#include "cylinder.h"  // Cylinder class definition

void virtualViaPointer( const Shape * );
void virtualViaReference( const Shape & );
```

fig10_20.cpp
(1 of 5)

Figure 5.22: **Cylinder** class implementation file. (part 2 of 2)

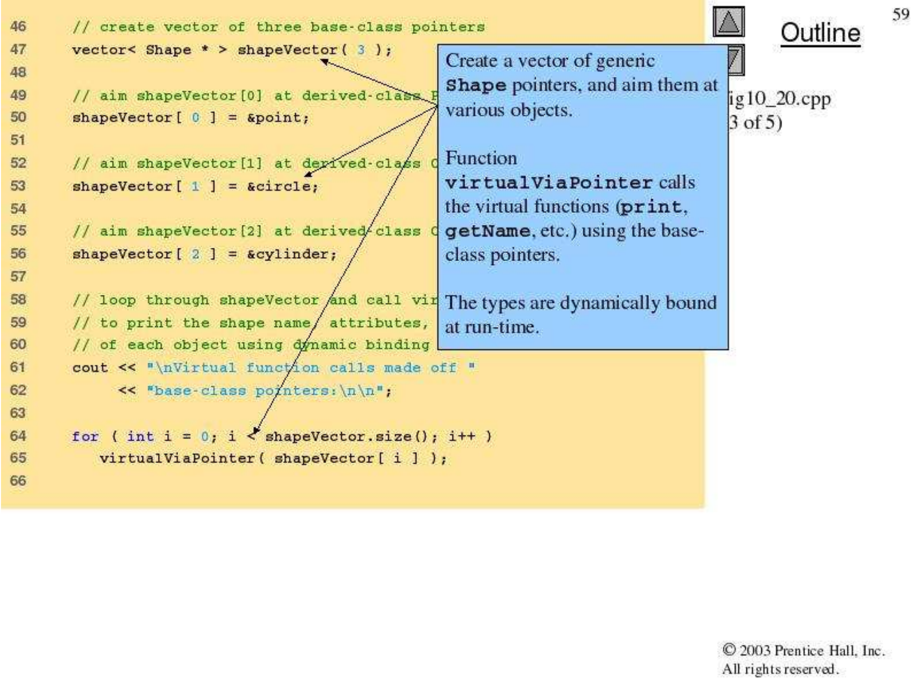


58

Outline

fig10_20.cpp
(2 of 5)

```
int main()
{
   // set floating-point number format
   cout << fixed << setprecision( 2 );

   Point point( 7, 11 );                   // create a Point
   Circle circle( 22, 8, 3.5 );            // create a Circle
   Cylinder cylinder( 10, 10, 3.3, 10 );  // create a Cylinder

   cout << point.getName() << ": ";    // static binding
   point.print();                      // static binding
   cout << '\n';

   cout << circle.getName() << ": ";   // static binding
   circle.print();                     // static binding
   cout << '\n';

   cout << cylinder.getName() << ": "; // static binding
   cylinder.print();                   // static binding
   cout << "\n\n";

```



59

Outline

ig10_20.cpp
3 of 5)

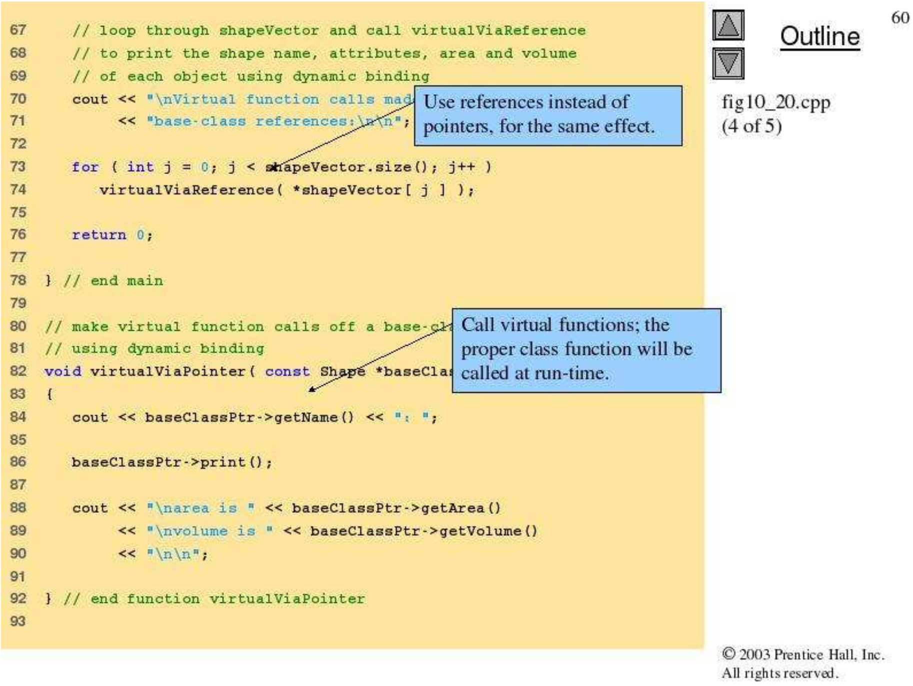
```
   // create vector of three base-class pointers
   vector< Shape * > shapeVector( 3 );

   // aim shapeVector[0] at derived-class P
   shapeVector[ 0 ] = &point;

   // aim shapeVector[1] at derived-class C
   shapeVector[ 1 ] = &circle;

   // aim shapeVector[2] at derived-class C
   shapeVector[ 2 ] = &cylinder;

   // loop through shapeVector and call vir
   // to print the shape name, attributes,
   // of each object using dynamic binding
   cout << "\nVirtual function calls made off "
        << "base-class pointers:\n\n";

   for ( int i = 0; i < shapeVector.size(); i++ )
      virtualViaPointer( shapeVector[ i ] );

```

Create a vector of generic **Shape** pointers, and aim them at various objects.

Function **virtualViaPointer** calls the virtual functions (**print**, **getName**, etc.) using the base-class pointers.

The types are dynamically bound at run-time.

Figure 5.23: Demonstarting polymorphism via a hierarchy headed by an abstract base class. (part 1 of 3)



```
   // loop through shapeVector and call virtualViaReference
   // to print the shape name, attributes, area and volume
   // of each object using dynamic binding
   cout << "\nVirtual function calls mad
        << "base-class references:\n\n";

   for ( int j = 0; j < shapeVector.size(); j++ )
      virtualViaReference( *shapeVector[ j ] );

   return 0;

} // end main

// make virtual function calls off a base-cl
// using dynamic binding
void virtualViaPointer( const Shape *baseCla
{
   cout << baseClassPtr->getName() << ": ";

   baseClassPtr->print();

   cout << "\narea is " << baseClassPtr->getArea()
        << "\nvolume is " << baseClassPtr->getVolume()
        << "\n\n";

} // end function virtualViaPointer
```

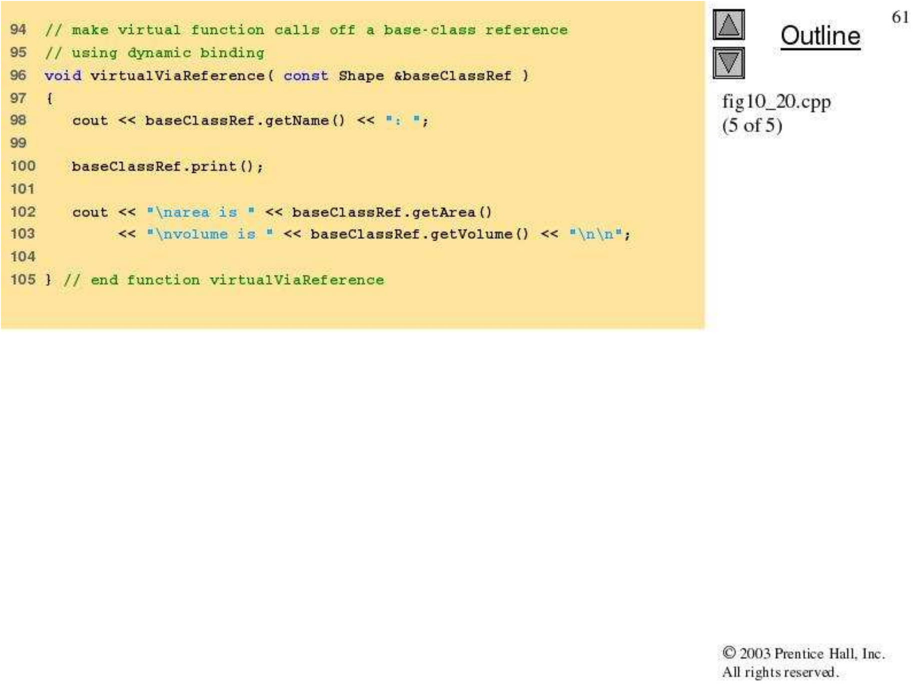
Use references instead of pointers, for the same effect.

Call virtual functions; the proper class function will be called at run-time.

60

Outline

fig10_20.cpp
(4 of 5)



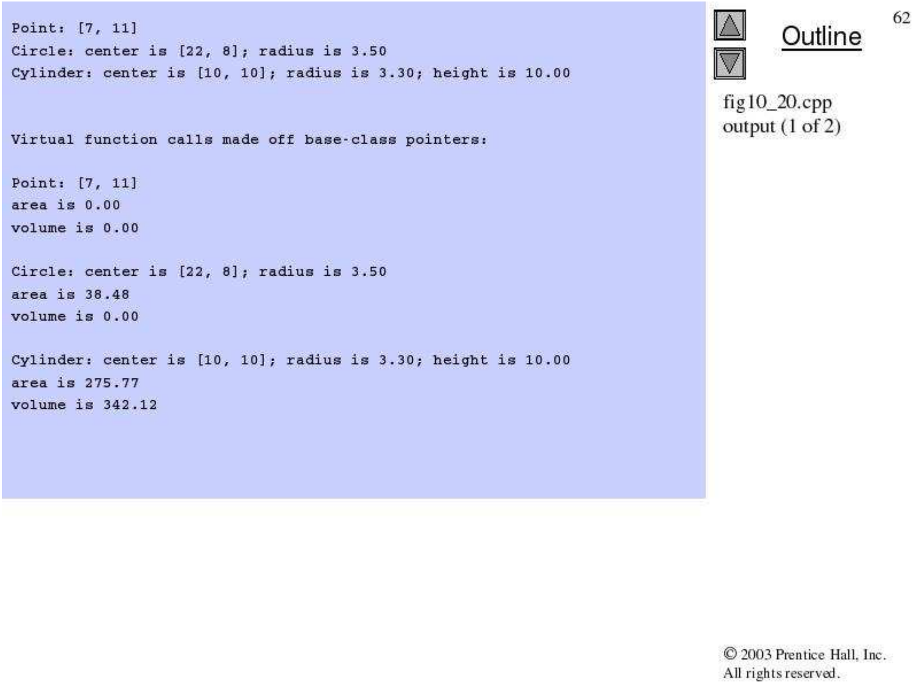
```
// make virtual function calls off a base-class reference
// using dynamic binding
void virtualViaReference( const Shape &baseClassRef )
{
   cout << baseClassRef.getName() << ": ";

   baseClassRef.print();

   cout << "\narea is " << baseClassRef.getArea()
        << "\nvolume is " << baseClassRef.getVolume() << "\n\n";

} // end function virtualViaReference
```

61

Outline

fig10_20.cpp
(5 of 5)

Figure 5.24: Demonstarting polymorphism via a hierarchy headed by an abstract base class. (part 2 of 3)

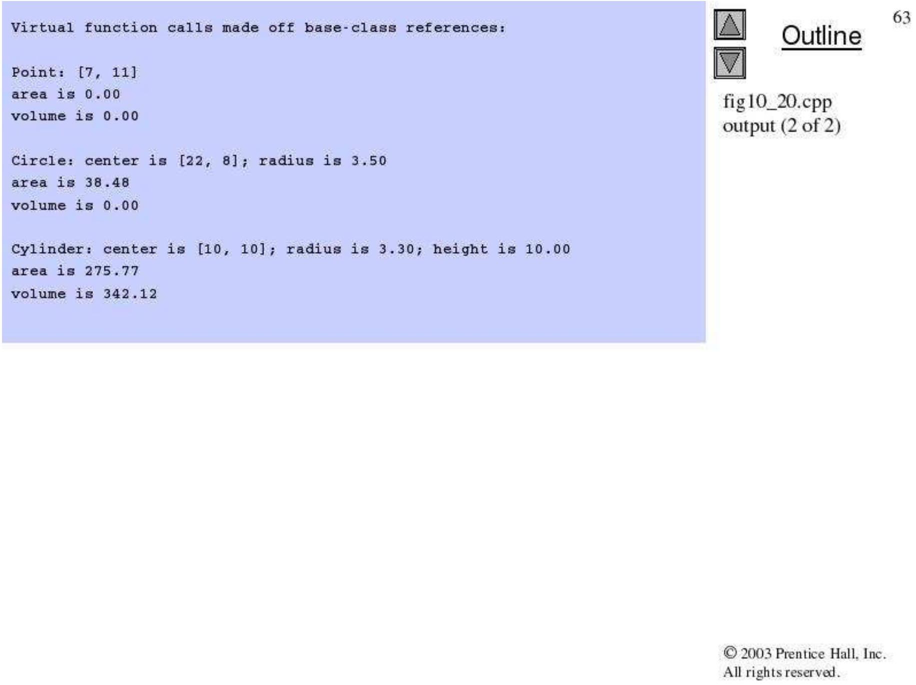


62

Outline

fig10_20.cpp
output (1 of 2)

```
Point: [7, 11]
Circle: center is [22, 8]; radius is 3.50
Cylinder: center is [10, 10]; radius is 3.30; height is 10.00


Virtual function calls made off base-class pointers:

Point: [7, 11]
area is 0.00
volume is 0.00

Circle: center is [22, 8]; radius is 3.50
area is 38.48
volume is 0.00

Cylinder: center is [10, 10]; radius is 3.30; height is 10.00
area is 275.77
volume is 342.12
```



63

Outline

fig10_20.cpp
output (2 of 2)

```
Virtual function calls made off base-class references:

Point: [7, 11]
area is 0.00
volume is 0.00

Circle: center is [22, 8]; radius is 3.50
area is 38.48
volume is 0.00

Cylinder: center is [10, 10]; radius is 3.30; height is 10.00
area is 275.77
volume is 342.12
```

Figure 5.25: Demonstarting polymorphism via a hierarchy headed by an abstract base class. (part 3 of 3)

5.7 Polymorphism, Virtual Functions and Dynamic Binding "Under the Hood"

- Polymorphism has overhead
  - Not used in STL (Standard Template Library) to optimize performance
- virtual function table (vtable)
  - Every class with a **virtual** function has a vtable
  - For every **virtual** function, vtable has pointer to the proper function
  - If derived class has same function as base class; function pointer aims at base-class function
  - Detailed explanation in Fig. 10.21 (in book) (will not be covered)

5.8 Virtual Destructors

- Base class pointer to derived object; if destroyed using **delete**, behavior unspecified
- Simple fix
  - Declare base-class destructor virtual; makes derived-class destructors virtual
  - Now, when **delete** used appropriate destructor called
- When derived-class object destroyed
  - Derived-class destructor executes first
  - Base-class destructor executes afterwards
- Constructors cannot be virtual

5.9 Case Study: Payroll System Using Polymorphism

- Base class Employee

  - Pure virtual function **earnings** (returns pay)
    - Pure virtual because need to know employee type
    - Cannot calculate for generic employee
  - Other classes derive from **Employee**

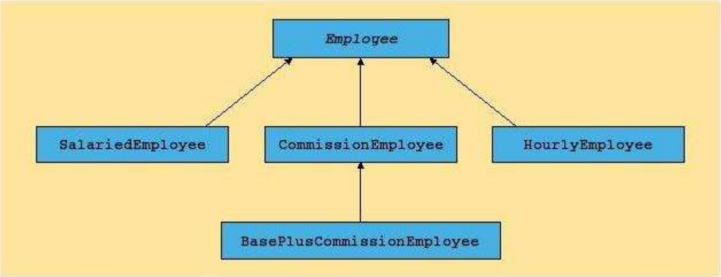
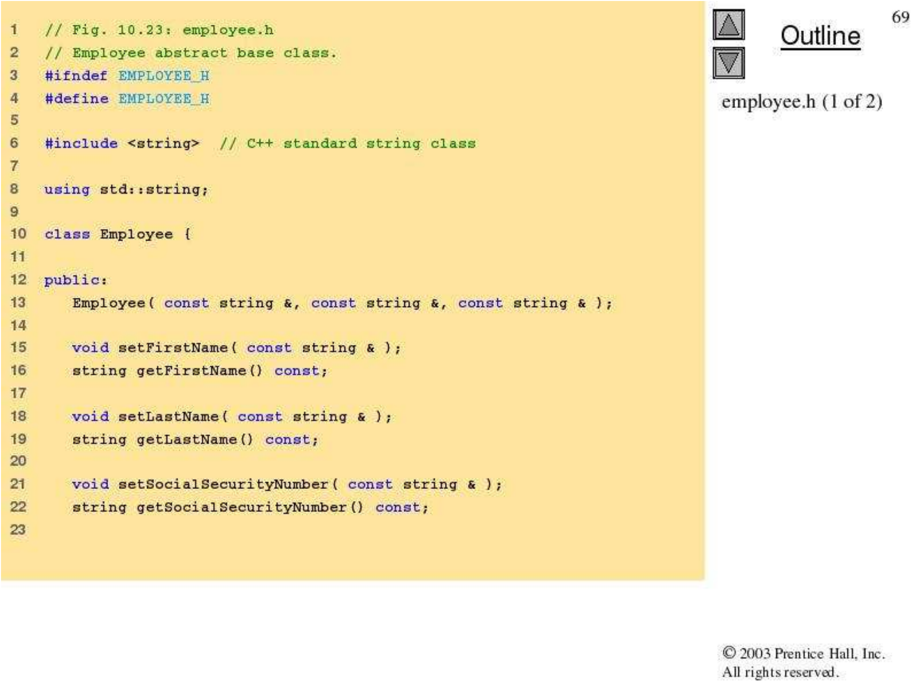


Figure 5.26: Class hierarchy for the polymorphic employee-payroll application.

- Downcasting
  - **dynamic_cast** operator
    - Determine object's type at runtime
    - Returns 0 if not of proper type (cannot be cast)
    - **NewClass *ptr = dynamic_cast ¡ NewClass *¿ objectPtr;**
- Keyword typeid
  - Header **¡typeinfo¿**
  - Usage: **typeid(object)**
    - Returns **type_info** object
    - Has information about type of operand, including name
    - **typeid(object).name()**



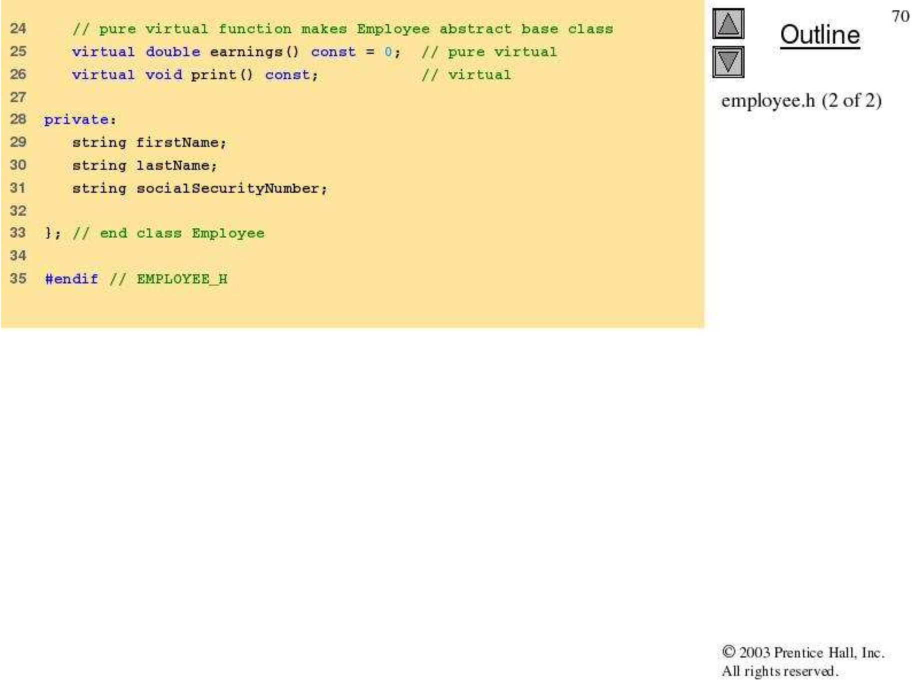
```
// Fig. 10.23: employee.h
// Employee abstract base class.
#ifndef EMPLOYEE_H
#define EMPLOYEE_H

#include <string>  // C++ standard string class

using std::string;

class Employee {

public:
   Employee( const string &, const string &, const string & );

   void setFirstName( const string & );
   string getFirstName() const;

   void setLastName( const string & );
   string getLastName() const;

   void setSocialSecurityNumber( const string & );
   string getSocialSecurityNumber() const;

```

Outline

69

employee.h (1 of 2)

© 2003 Prentice Hall, Inc.
All rights reserved.



```
   // pure virtual function makes Employee abstract base class
   virtual double earnings() const = 0;  // pure virtual
   virtual void print() const;           // virtual

private:
   string firstName;
   string lastName;
   string socialSecurityNumber;

}; // end class Employee

#endif // EMPLOYEE_H
```

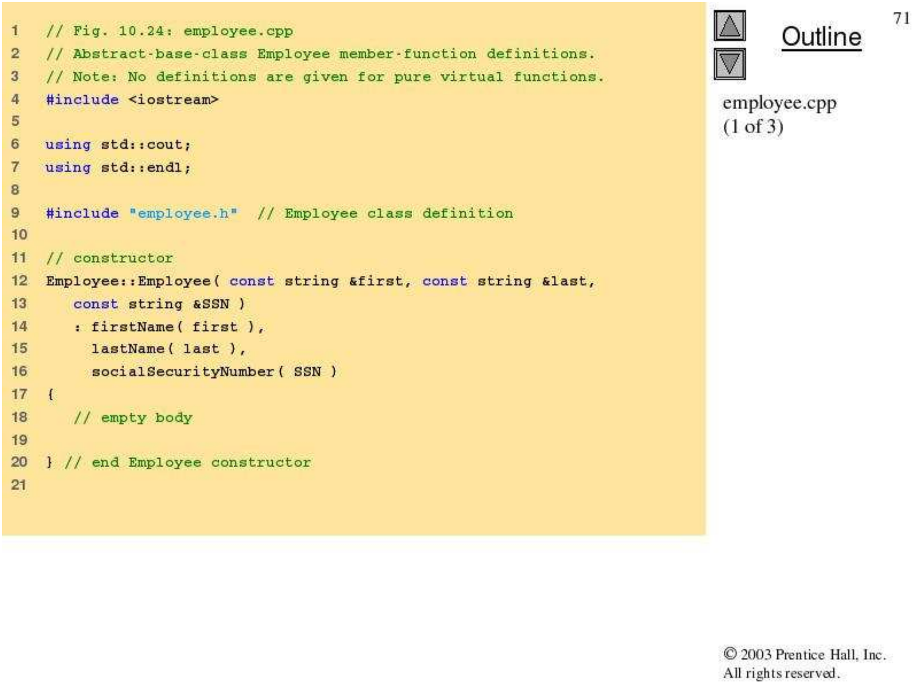
Outline

70

employee.h (2 of 2)

© 2003 Prentice Hall, Inc.
All rights reserved.

Figure 5.27: **Employee** class header file.



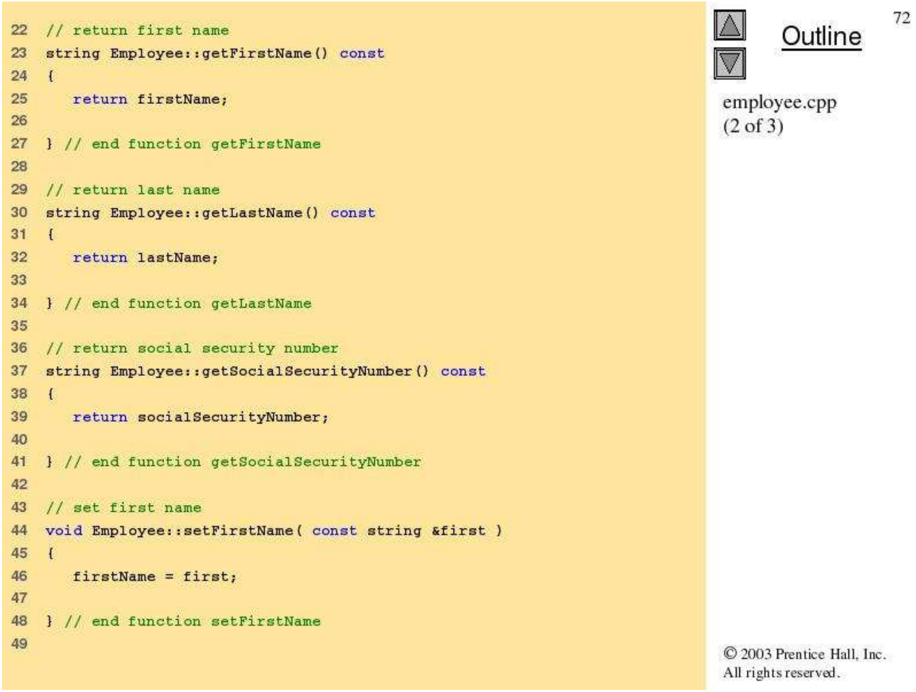
71

Outline

employee.cpp
(1 of 3)

```
// Fig. 10.24: employee.cpp
// Abstract-base-class Employee member-function definitions.
// Note: No definitions are given for pure virtual functions.
#include <iostream>

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

#include "employee.h"  // Employee class definition

// constructor
Employee::Employee( const string &first, const string &last,
   const string &SSN )
   : firstName( first ),
     lastName( last ),
     socialSecurityNumber( SSN )
{
   // empty body

} // end Employee constructor

```

© 2003 Prentice Hall, Inc.
All rights reserved.



72

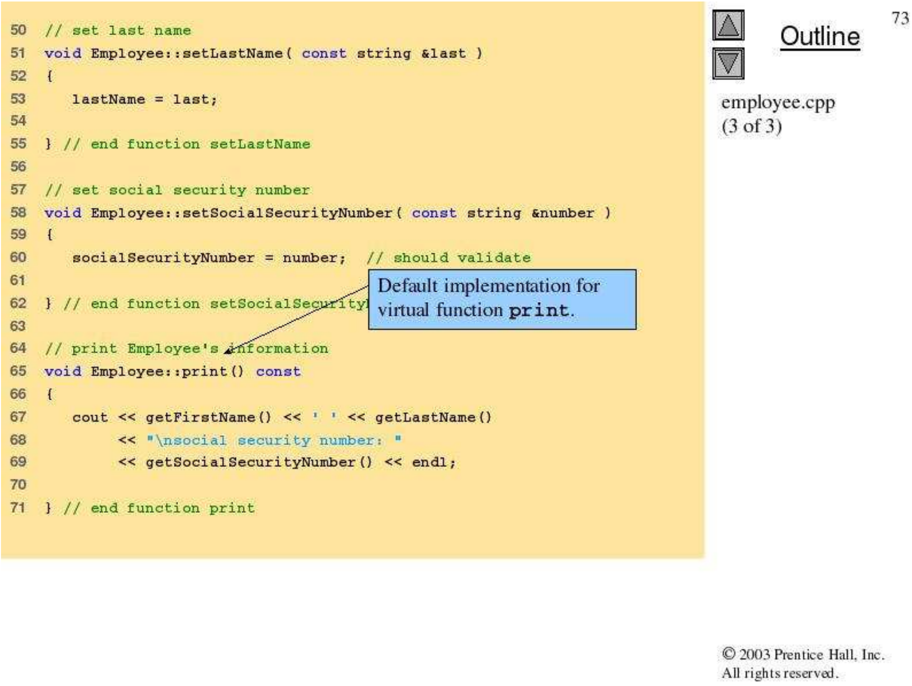
Outline

employee.cpp
(2 of 3)

```
// return first name
string Employee::getFirstName() const
{
   return firstName;

} // end function getFirstName

// return last name
string Employee::getLastName() const
{
   return lastName;

} // end function getLastName

// return social security number
string Employee::getSocialSecurityNumber() const
{
   return socialSecurityNumber;

} // end function getSocialSecurityNumber

// set first name
void Employee::setFirstName( const string &first )
{
   firstName = first;

} // end function setFirstName

```

© 2003 Prentice Hall, Inc.
All rights reserved.

Figure 5.28: **Employee** class implementation file. (part 1 of 2)



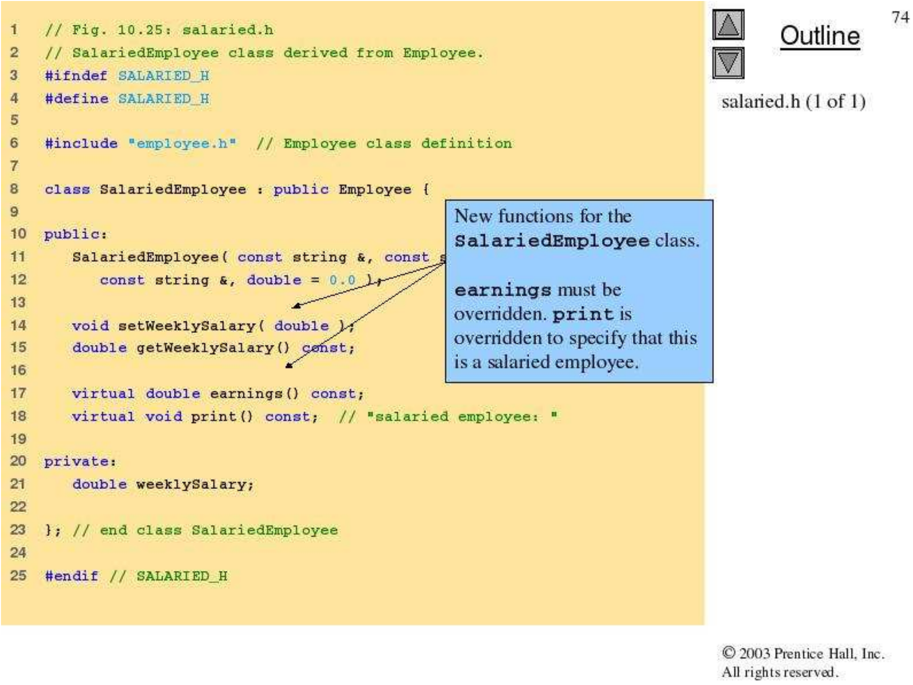
73

Outline

employee.cpp
(3 of 3)

```
// set last name
void Employee::setLastName( const string &last )
{
   lastName = last;

} // end function setLastName

// set social security number
void Employee::setSocialSecurityNumber( const string &number )
{
   socialSecurityNumber = number;  // should validate

} // end function setSocialSecurity

// print Employee's information
void Employee::print() const
{
   cout << getFirstName() << ' ' << getLastName()
        << "\nsocial security number: "
        << getSocialSecurityNumber() << endl;

} // end function print
```

Default implementation for virtual function **print**.



74

Outline

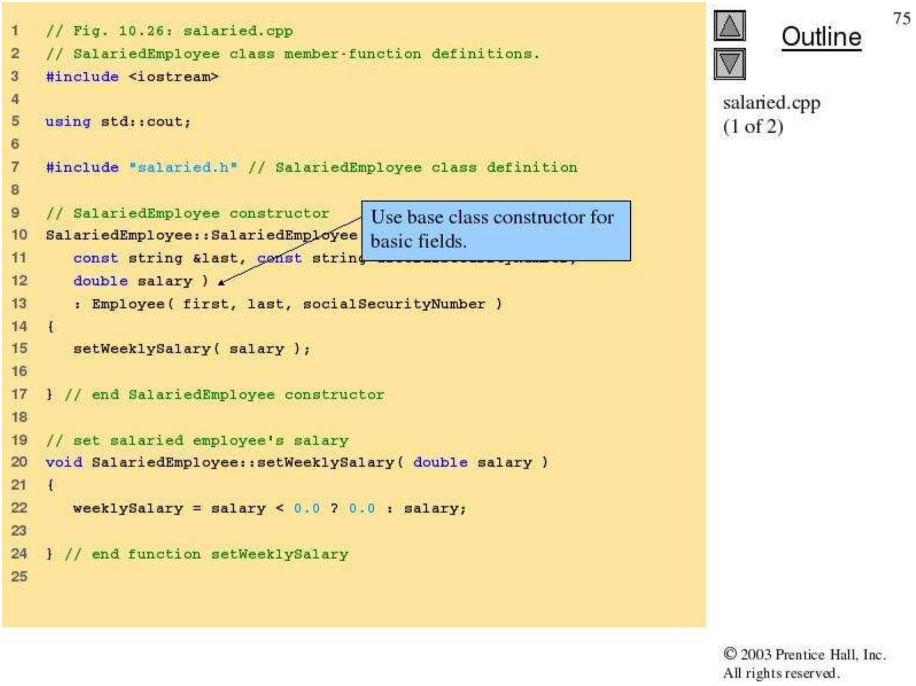
salaried.h (1 of 1)

```
// Fig. 10.25: salaried.h
// SalariedEmployee class derived from Employee.
#ifndef SALARIED_H
#define SALARIED_H

#include "employee.h"  // Employee class definition

class SalariedEmployee : public Employee {

public:
   SalariedEmployee( const string &, const s
      const string &, double = 0.0 );

   void setWeeklySalary( double );
   double getWeeklySalary() const;

   virtual double earnings() const;
   virtual void print() const;  // "salaried employee: "

private:
   double weeklySalary;

}; // end class SalariedEmployee

#endif // SALARIED_H
```

New functions for the **SalariedEmployee** class.

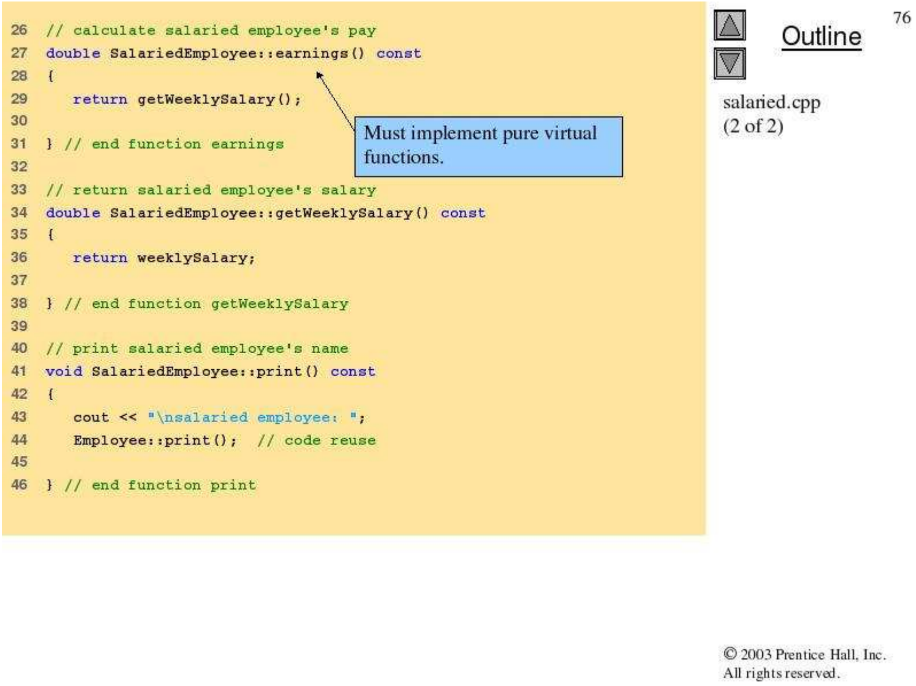
earnings must be overridden. **print** is overridden to specify that this is a salaried employee.

Figure 5.29: **Employee** class implementation file (part 2 of 2) and **SalariedEmployee** class header file.



```
// Fig. 10.26: salaried.cpp
// SalariedEmployee class member-function definitions.
#include <iostream>

using std::cout;

#include "salaried.h" // SalariedEmployee class definition

// SalariedEmployee constructor
SalariedEmployee::SalariedEmployee
   const string &last, const string
   double salary )
   : Employee( first, last, socialSecurityNumber )
{
   setWeeklySalary( salary );

} // end SalariedEmployee constructor

// set salaried employee's salary
void SalariedEmployee::setWeeklySalary( double salary )
{
   weeklySalary = salary < 0.0 ? 0.0 : salary;

} // end function setWeeklySalary

```

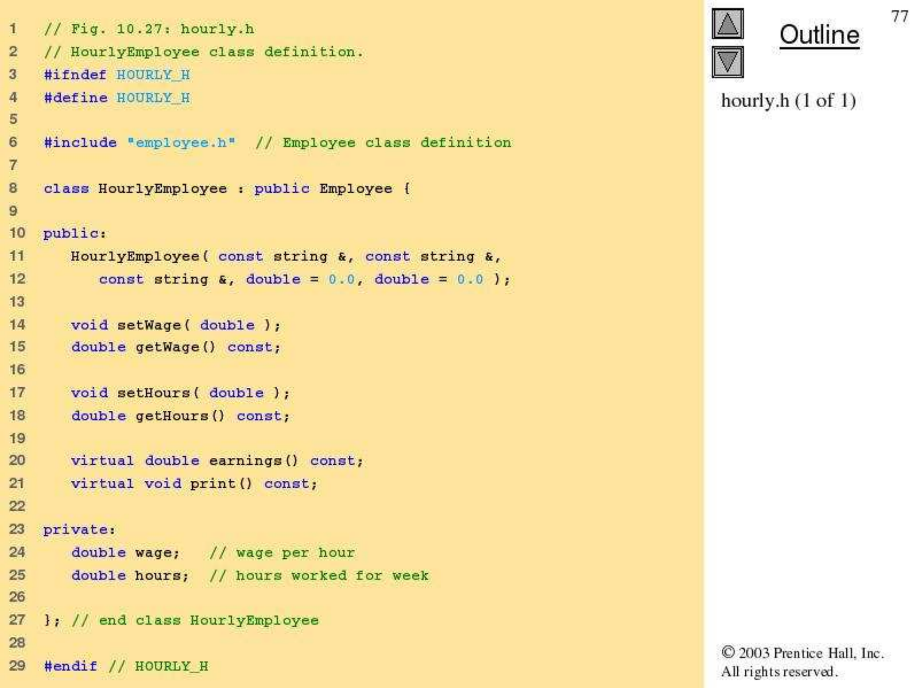
Use base class constructor for basic fields.



```
// calculate salaried employee's pay
double SalariedEmployee::earnings() const
{
   return getWeeklySalary();

} // end function earnings

// return salaried employee's salary
double SalariedEmployee::getWeeklySalary() const
{
   return weeklySalary;

} // end function getWeeklySalary

// print salaried employee's name
void SalariedEmployee::print() const
{
   cout << "\nsalaried employee: ";
   Employee::print();  // code reuse

} // end function print
```

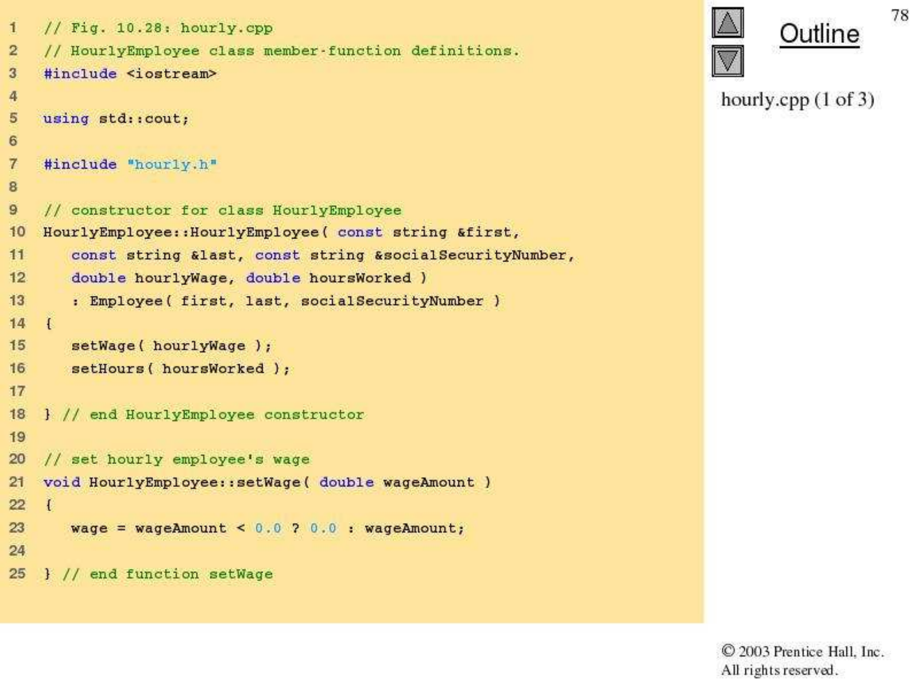
Must implement pure virtual functions.

Figure 5.30: **SalariedEmployee** class implementation file.



```cpp
// Fig. 10.27: hourly.h
// HourlyEmployee class definition.
#ifndef HOURLY_H
#define HOURLY_H

#include "employee.h"  // Employee class definition

class HourlyEmployee : public Employee {

public:
   HourlyEmployee( const string &, const string &,
      const string &, double = 0.0, double = 0.0 );

   void setWage( double );
   double getWage() const;

   void setHours( double );
   double getHours() const;

   virtual double earnings() const;
   virtual void print() const;

private:
   double wage;   // wage per hour
   double hours;  // hours worked for week

}; // end class HourlyEmployee

#endif // HOURLY_H
```

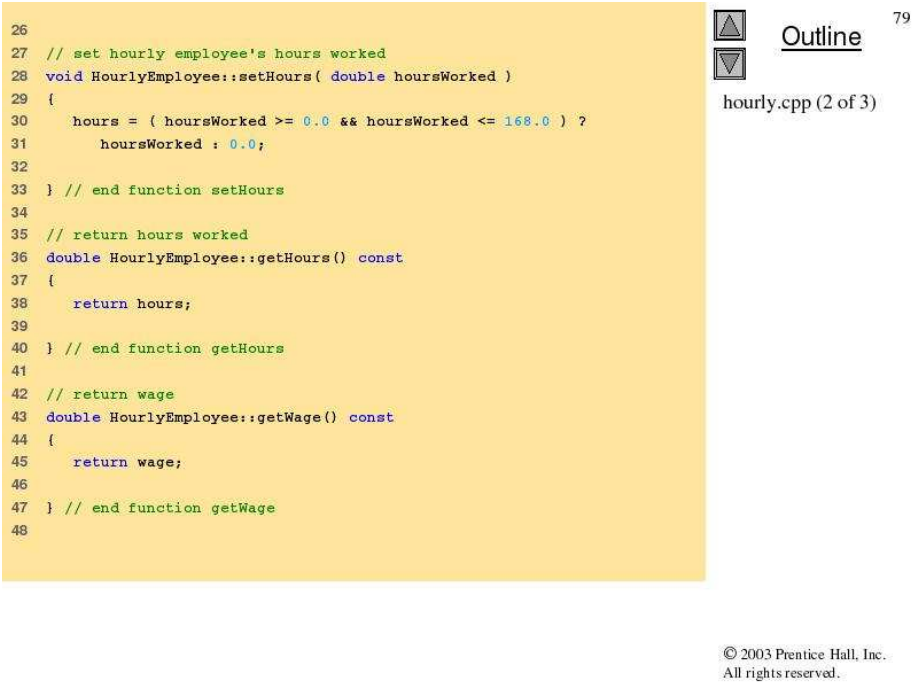
hourly.h (1 of 1)



```cpp
// Fig. 10.28: hourly.cpp
// HourlyEmployee class member-function definitions.
#include <iostream>

using std::cout;

#include "hourly.h"

// constructor for class HourlyEmployee
HourlyEmployee::HourlyEmployee( const string &first,
   const string &last, const string &socialSecurityNumber,
   double hourlyWage, double hoursWorked )
   : Employee( first, last, socialSecurityNumber )
{
   setWage( hourlyWage );
   setHours( hoursWorked );

} // end HourlyEmployee constructor

// set hourly employee's wage
void HourlyEmployee::setWage( double wageAmount )
{
   wage = wageAmount < 0.0 ? 0.0 : wageAmount;

} // end function setWage
```

hourly.cpp (1 of 3)

Figure 5.31: **HourlyEmployee** class header file.



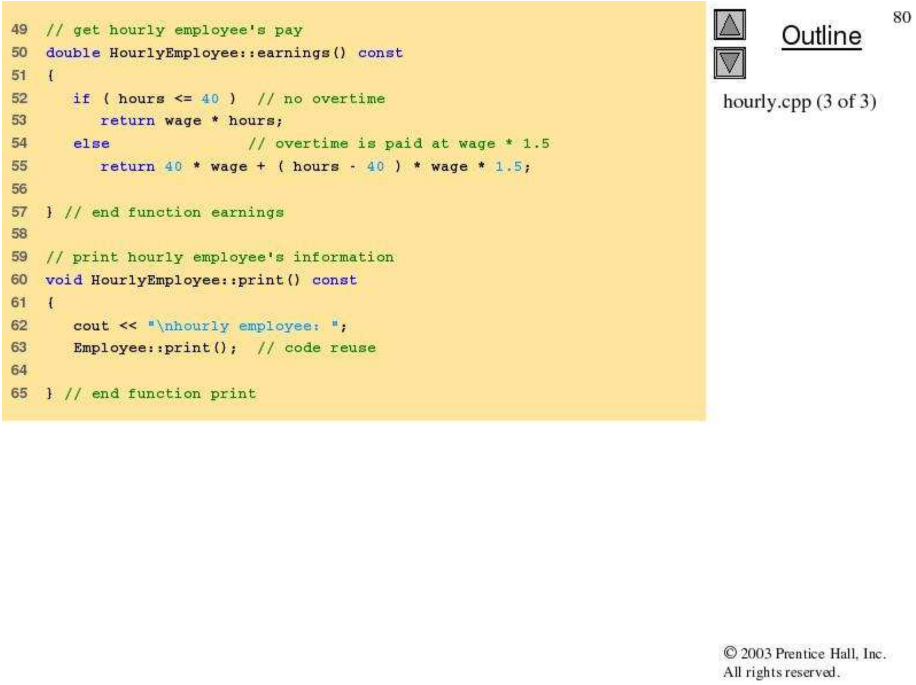
```

// set hourly employee's hours worked
void HourlyEmployee::setHours( double hoursWorked )
{
   hours = ( hoursWorked >= 0.0 && hoursWorked <= 168.0 ) ?
      hoursWorked : 0.0;

} // end function setHours

// return hours worked
double HourlyEmployee::getHours() const
{
   return hours;

} // end function getHours

// return wage
double HourlyEmployee::getWage() const
{
   return wage;

} // end function getWage

```

79

Outline

hourly.cpp (2 of 3)

© 2003 Prentice Hall, Inc.
All rights reserved.



```
// get hourly employee's pay
double HourlyEmployee::earnings() const
{
   if ( hours <= 40 )  // no overtime
      return wage * hours;
   else                // overtime is paid at wage * 1.5
      return 40 * wage + ( hours - 40 ) * wage * 1.5;

} // end function earnings

// print hourly employee's information
void HourlyEmployee::print() const
{
   cout << "\nhourly employee: ";
   Employee::print();  // code reuse

} // end function print
```

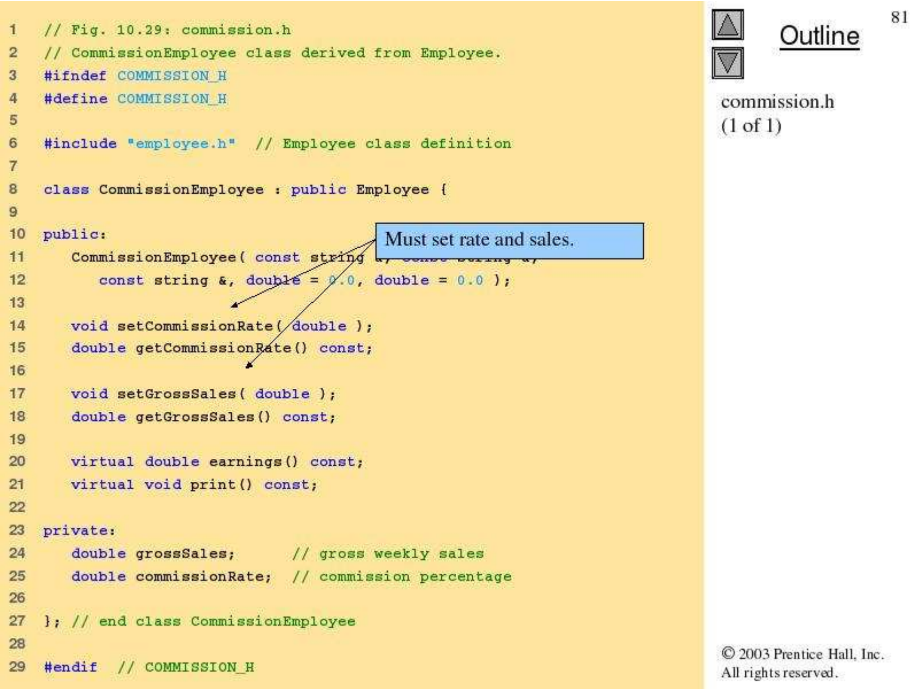
80

Outline

hourly.cpp (3 of 3)

© 2003 Prentice Hall, Inc.
All rights reserved.

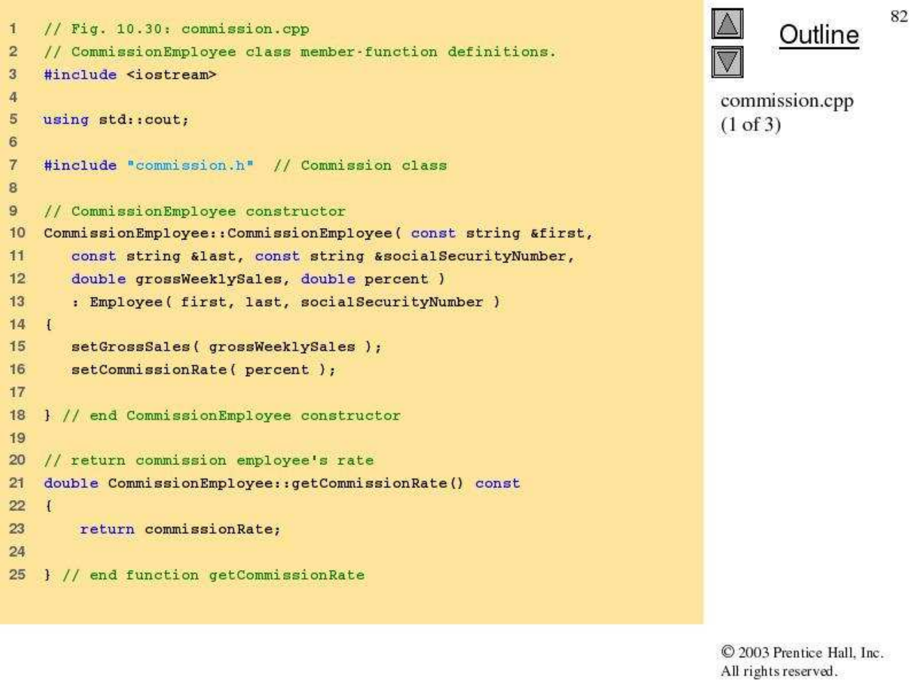
Figure 5.32: **HourlyEmployee** class implementation file.



```
// Fig. 10.29: commission.h
// CommissionEmployee class derived from Employee.
#ifndef COMMISSION_H
#define COMMISSION_H

#include "employee.h"  // Employee class definition

class CommissionEmployee : public Employee {

public:
   CommissionEmployee( const string &, const string &,
      const string &, double = 0.0, double = 0.0 );

   void setCommissionRate( double );
   double getCommissionRate() const;

   void setGrossSales( double );
   double getGrossSales() const;

   virtual double earnings() const;
   virtual void print() const;

private:
   double grossSales;      // gross weekly sales
   double commissionRate;  // commission percentage

}; // end class CommissionEmployee

#endif  // COMMISSION_H
```

Must set rate and sales.

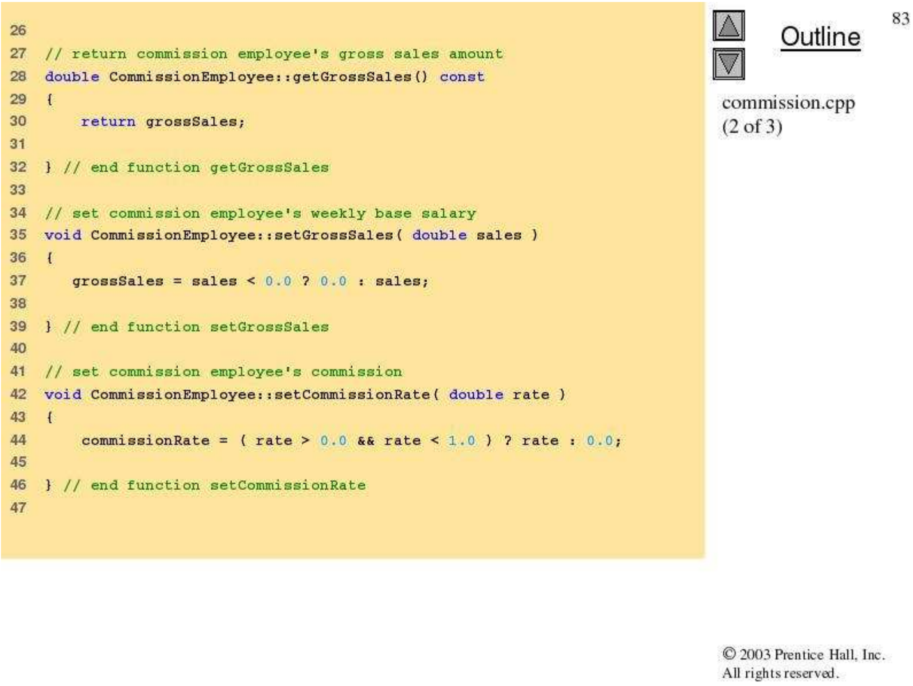
commission.h
(1 of 1)



```
// Fig. 10.30: commission.cpp
// CommissionEmployee class member-function definitions.
#include <iostream>

using std::cout;

#include "commission.h"  // Commission class

// CommissionEmployee constructor
CommissionEmployee::CommissionEmployee( const string &first,
   const string &last, const string &socialSecurityNumber,
   double grossWeeklySales, double percent )
   : Employee( first, last, socialSecurityNumber )
{
   setGrossSales( grossWeeklySales );
   setCommissionRate( percent );

} // end CommissionEmployee constructor

// return commission employee's rate
double CommissionEmployee::getCommissionRate() const
{
    return commissionRate;

} // end function getCommissionRate
```

commission.cpp
(1 of 3)

Figure 5.33: **CommissionEmployee** class header file.



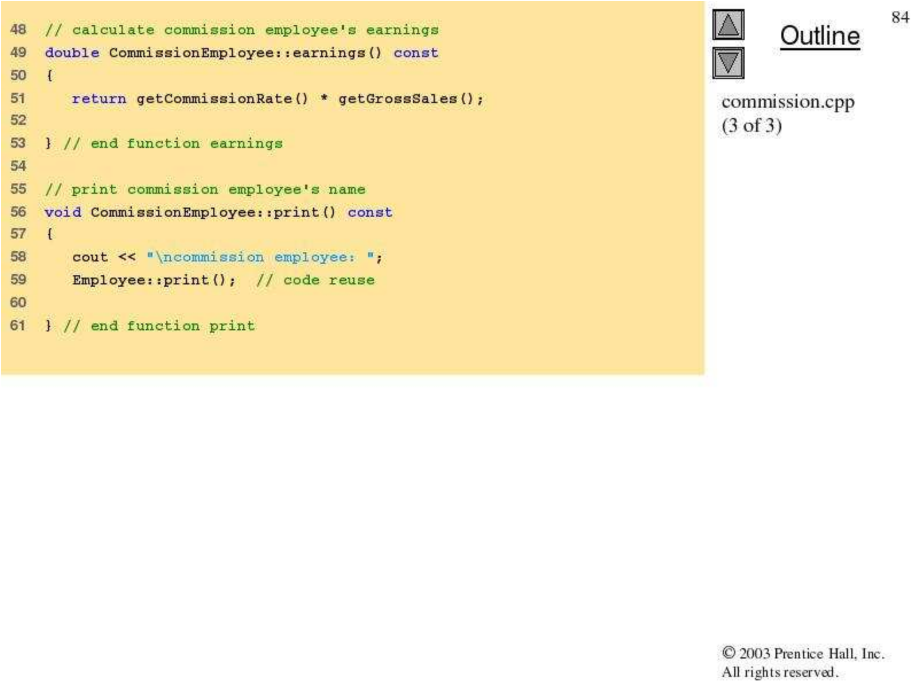
```

// return commission employee's gross sales amount
double CommissionEmployee::getGrossSales() const
{
    return grossSales;

} // end function getGrossSales

// set commission employee's weekly base salary
void CommissionEmployee::setGrossSales( double sales )
{
   grossSales = sales < 0.0 ? 0.0 : sales;

} // end function setGrossSales

// set commission employee's commission
void CommissionEmployee::setCommissionRate( double rate )
{
    commissionRate = ( rate > 0.0 && rate < 1.0 ) ? rate : 0.0;

} // end function setCommissionRate

```

83

Outline

commission.cpp
(2 of 3)

© 2003 Prentice Hall, Inc.
All rights reserved.



```
// calculate commission employee's earnings
double CommissionEmployee::earnings() const
{
   return getCommissionRate() * getGrossSales();

} // end function earnings

// print commission employee's name
void CommissionEmployee::print() const
{
   cout << "\ncommission employee: ";
   Employee::print();  // code reuse

} // end function print
```

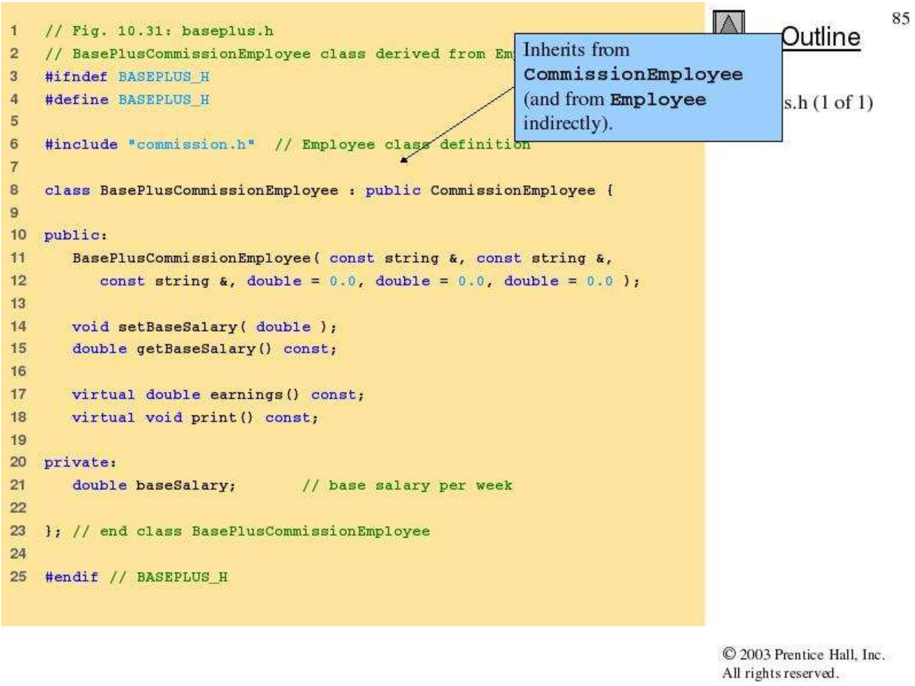
84

Outline

commission.cpp
(3 of 3)

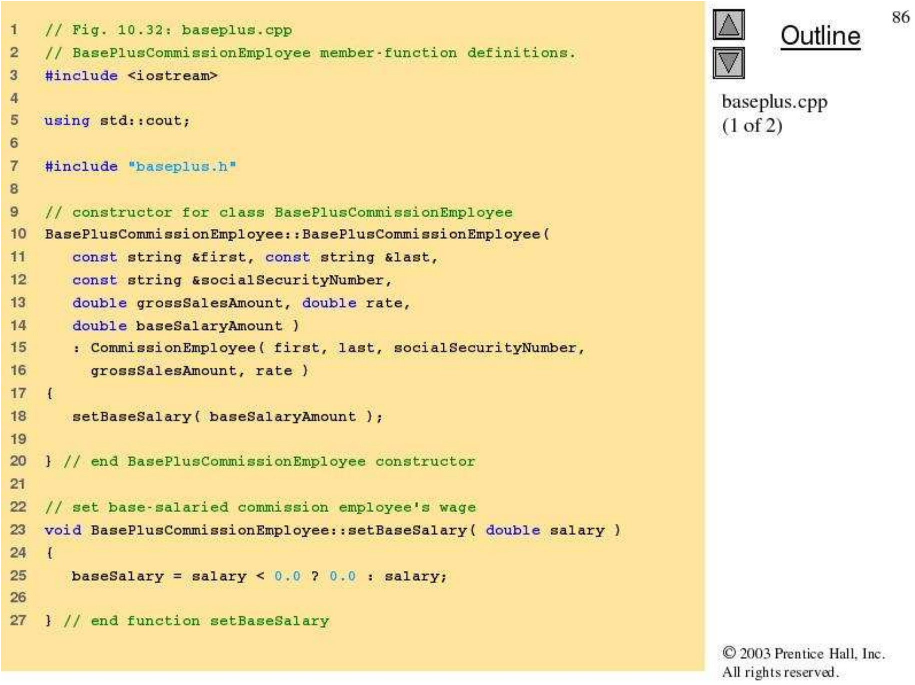
© 2003 Prentice Hall, Inc.
All rights reserved.

Figure 5.34: **CommissionEmployee** class implementation file.



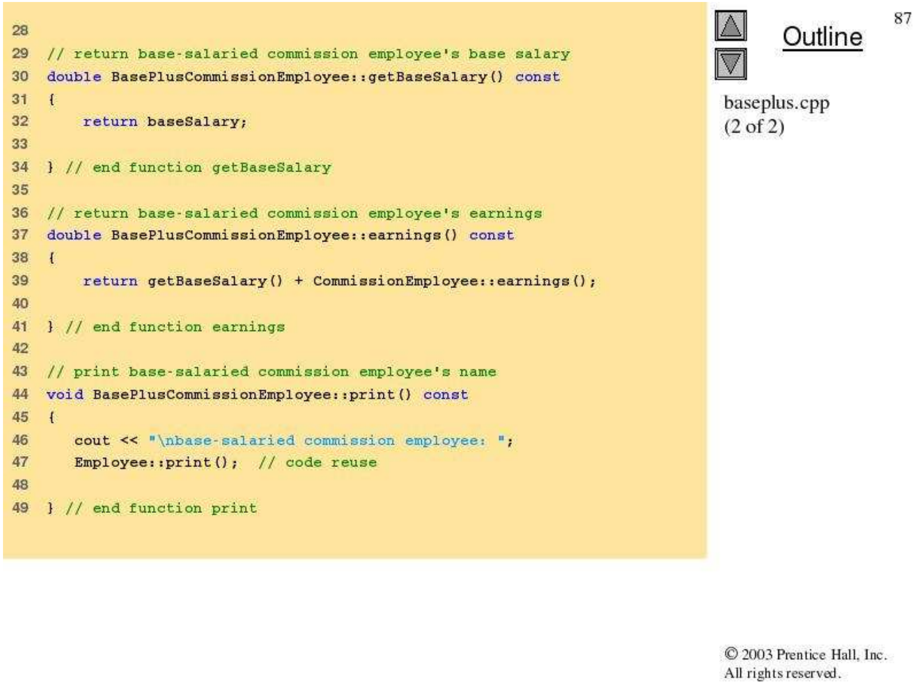
```
// Fig. 10.31: baseplus.h
// BasePlusCommissionEmployee class derived from Em
#ifndef BASEPLUS_H
#define BASEPLUS_H

#include "commission.h"  // Employee class definition

class BasePlusCommissionEmployee : public CommissionEmployee {

public:
   BasePlusCommissionEmployee( const string &, const string &,
      const string &, double = 0.0, double = 0.0, double = 0.0 );

   void setBaseSalary( double );
   double getBaseSalary() const;

   virtual double earnings() const;
   virtual void print() const;

private:
   double baseSalary;       // base salary per week

}; // end class BasePlusCommissionEmployee

#endif // BASEPLUS_H
```

Inherits from **CommissionEmployee** (and from **Employee** indirectly).



```
// Fig. 10.32: baseplus.cpp
// BasePlusCommissionEmployee member-function definitions.
#include <iostream>

using std::cout;

#include "baseplus.h"

// constructor for class BasePlusCommissionEmployee
BasePlusCommissionEmployee::BasePlusCommissionEmployee(
   const string &first, const string &last,
   const string &socialSecurityNumber,
   double grossSalesAmount, double rate,
   double baseSalaryAmount )
   : CommissionEmployee( first, last, socialSecurityNumber,
     grossSalesAmount, rate )
{
   setBaseSalary( baseSalaryAmount );

} // end BasePlusCommissionEmployee constructor

// set base-salaried commission employee's wage
void BasePlusCommissionEmployee::setBaseSalary( double salary )
{
   baseSalary = salary < 0.0 ? 0.0 : salary;

} // end function setBaseSalary
```

Figure 5.35: **BasePlusCommissionEmployee** class header file.

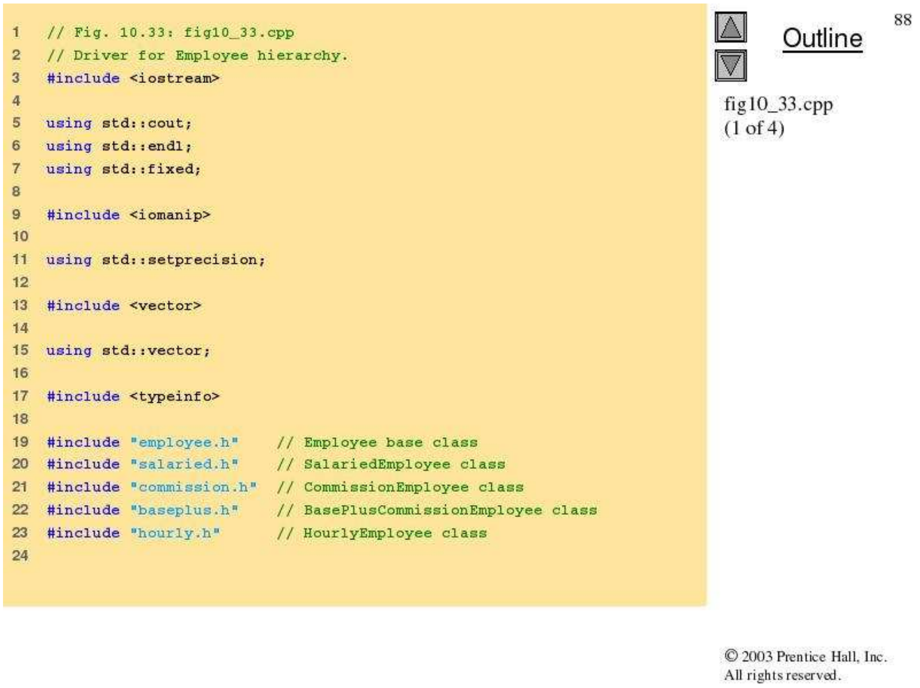


87

Outline

baseplus.cpp
(2 of 2)

```

// return base-salaried commission employee's base salary
double BasePlusCommissionEmployee::getBaseSalary() const
{
    return baseSalary;

} // end function getBaseSalary

// return base-salaried commission employee's earnings
double BasePlusCommissionEmployee::earnings() const
{
    return getBaseSalary() + CommissionEmployee::earnings();

} // end function earnings

// print base-salaried commission employee's name
void BasePlusCommissionEmployee::print() const
{
   cout << "\nbase-salaried commission employee: ";
   Employee::print();  // code reuse

} // end function print
```



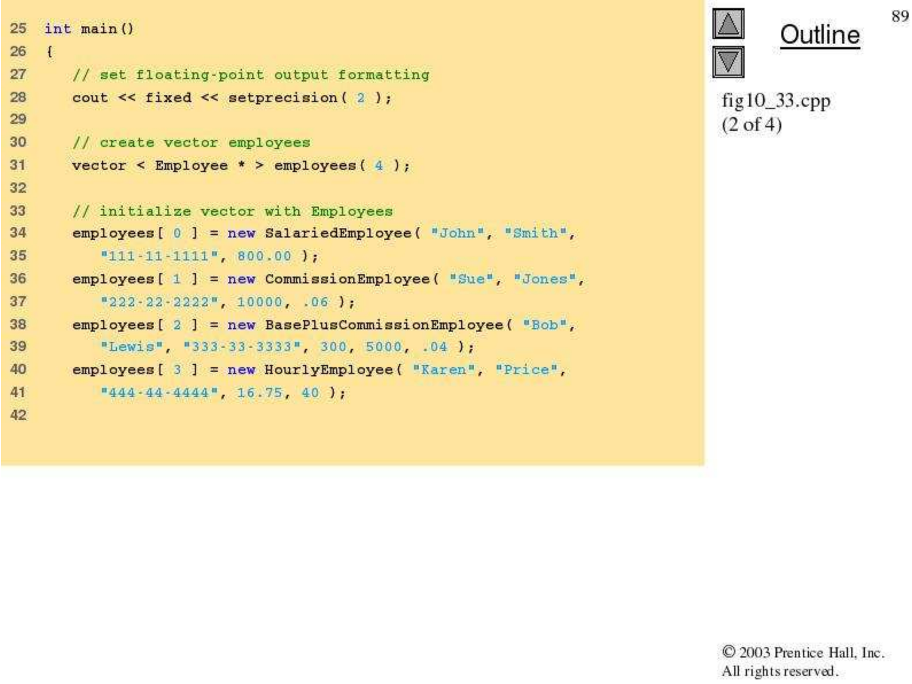
88

Outline

fig10_33.cpp
(1 of 4)

```
// Fig. 10.33: fig10_33.cpp
// Driver for Employee hierarchy.
#include <iostream>

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

#include <iomanip>

using std::setprecision;

#include <vector>

using std::vector;

#include <typeinfo>

#include "employee.h"    // Employee base class
#include "salaried.h"    // SalariedEmployee class
#include "commission.h"  // CommissionEmployee class
#include "baseplus.h"    // BasePlusCommissionEmployee class
#include "hourly.h"      // HourlyEmployee class

```

Figure 5.36: **BasePlusCommissionEmployee** class implementation file.



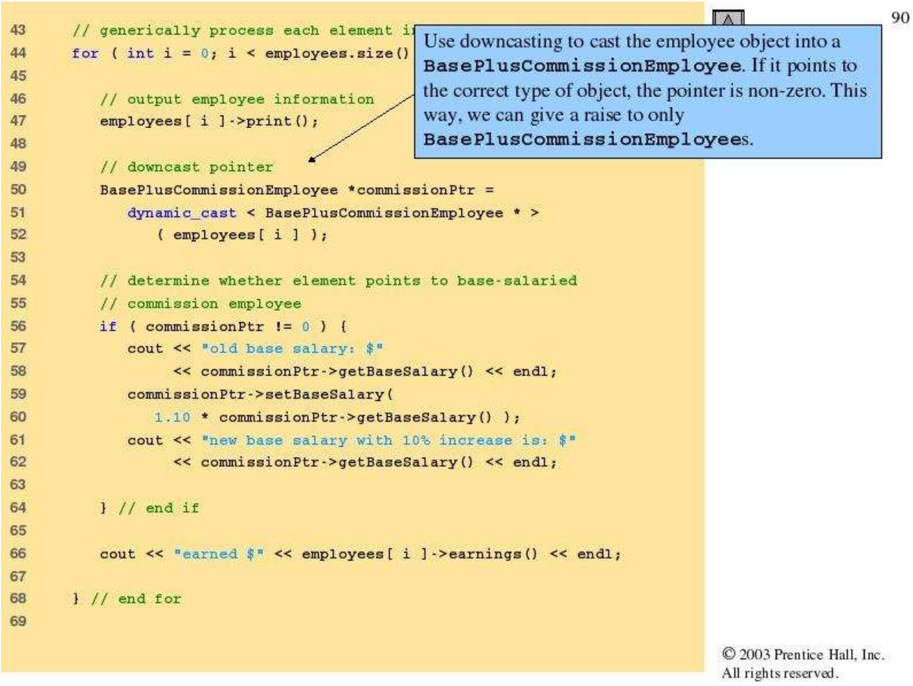
```
int main()
{
   // set floating-point output formatting
   cout << fixed << setprecision( 2 );

   // create vector employees
   vector < Employee * > employees( 4 );

   // initialize vector with Employees
   employees[ 0 ] = new SalariedEmployee( "John", "Smith",
      "111-11-1111", 800.00 );
   employees[ 1 ] = new CommissionEmployee( "Sue", "Jones",
      "222-22-2222", 10000, .06 );
   employees[ 2 ] = new BasePlusCommissionEmployee( "Bob",
      "Lewis", "333-33-3333", 300, 5000, .04 );
   employees[ 3 ] = new HourlyEmployee( "Karen", "Price",
      "444-44-4444", 16.75, 40 );

```

Outline

89

fig10_33.cpp
(2 of 4)

© 2003 Prentice Hall, Inc.
All rights reserved.



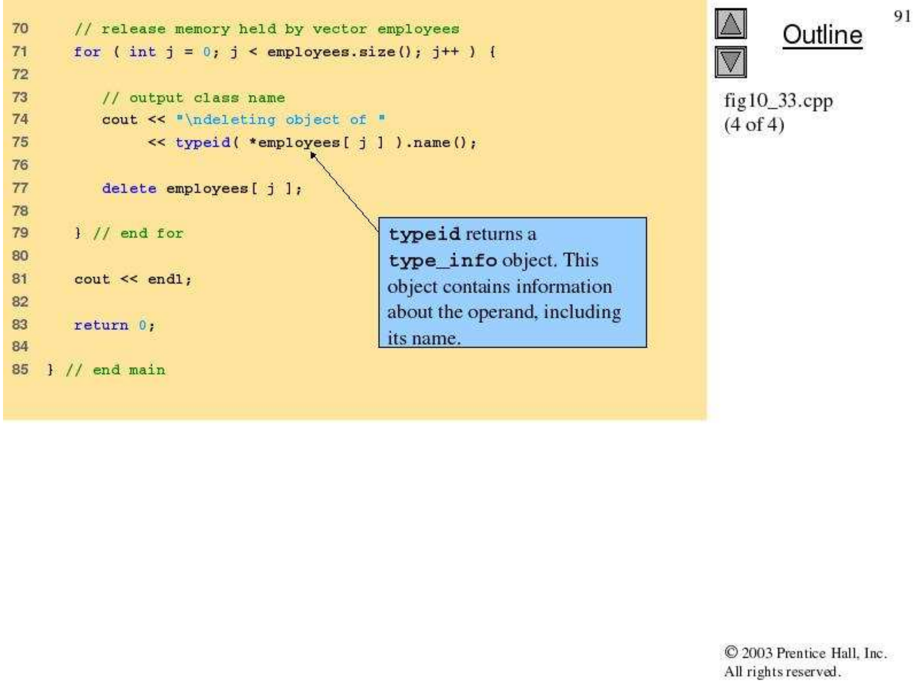
```
   // generically process each element i
   for ( int i = 0; i < employees.size()

      // output employee information
      employees[ i ]->print();

      // downcast pointer
      BasePlusCommissionEmployee *commissionPtr =
         dynamic_cast < BasePlusCommissionEmployee * >
            ( employees[ i ] );

      // determine whether element points to base-salaried
      // commission employee
      if ( commissionPtr != 0 ) {
         cout << "old base salary: $"
               << commissionPtr->getBaseSalary() << endl;
         commissionPtr->setBaseSalary(
            1.10 * commissionPtr->getBaseSalary() );
         cout << "new base salary with 10% increase is: $"
               << commissionPtr->getBaseSalary() << endl;

      } // end if

      cout << "earned $" << employees[ i ]->earnings() << endl;

   } // end for

```

Use downcasting to cast the employee object into a **BasePlusCommissionEmployee**. If it points to the correct type of object, the pointer is non-zero. This way, we can give a raise to only **BasePlusCommissionEmployee**s.

90

© 2003 Prentice Hall, Inc.
All rights reserved.

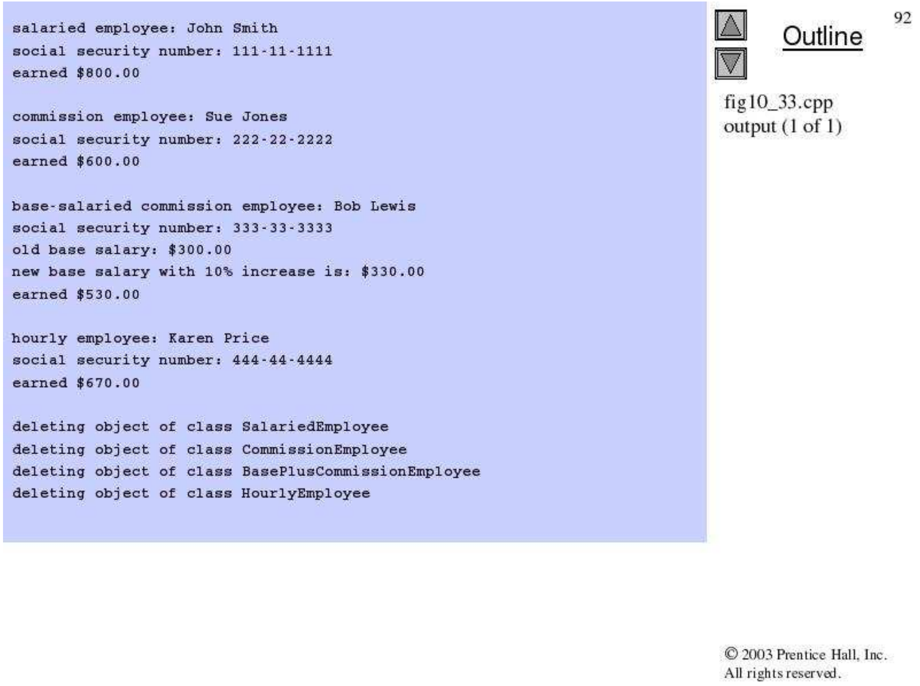
Figure 5.37: **Employee** class hierarchy driver program.(part 1 of 2)



```
   // release memory held by vector employees
   for ( int j = 0; j < employees.size(); j++ ) {

      // output class name
      cout << "\ndeleting object of "
           << typeid( *employees[ j ] ).name();

      delete employees[ j ];

   } // end for

   cout << endl;

   return 0;

} // end main
```

`typeid` returns a **`type_info`** object. This object contains information about the operand, including its name.

fig10_33.cpp
(4 of 4)



```
salaried employee: John Smith
social security number: 111-11-1111
earned $800.00

commission employee: Sue Jones
social security number: 222-22-2222
earned $600.00

base-salaried commission employee: Bob Lewis
social security number: 333-33-3333
old base salary: $300.00
new base salary with 10% increase is: $330.00
earned $530.00

hourly employee: Karen Price
social security number: 444-44-4444
earned $670.00

deleting object of class SalariedEmployee
deleting object of class CommissionEmployee
deleting object of class BasePlusCommissionEmployee
deleting object of class HourlyEmployee
```

fig10_33.cpp
output (1 of 1)

Figure 5.38: **Employee** class hierarchy driver program.(part 2 of 2)

5.10 vita

Cem Özdoğan was born in Merzifon, Amasya on October 23, 1969. He received his B.S. degree in Physics from the Middle East Technical University in June 1994. He received his M.S. degree in Physics from the Middle East Technical University in June 1996. He received his Ph.D. degree in Physics from the Middle East Technical University in June 2002. He worked as a research assistant from 1994 to 2001 in the department of physics, Kırıkkale University and Middle East Technical University and as instructor in the department of computer engineering, Çankaya University from 2001 to 2002. He is currently employed as Assist. Prof. in the department of computer engineering, Çankaya University. His main areas of interest are electronic structure calculations, parallel computing and scientific computing.