11.0

Class parameters, const

Parameter passing efficiency

A call-by-value parameter less efficient than a call-by-reference parameter

- The parameter is a local variable initialized to the value of the argument
  - This results in two copies of the argument
  - For vectors and strings, it means dynamic allocation and copying the values. Expensive!!

A call-by-reference parameter is more efficient

- The parameter is a placeholder replaced by the argument
  - There is only one copy of the argument

Class Parameters

- It can be much more efficient to use call-by-reference parameters when the parameter is of a class type
- When using a call-by-reference parameter
  - If the function does not change the value of the parameter, mark the parameter so the compiler knows it should not be changed

const Parameter Modifier

- To mark a call-by-reference parameter so it cannot be changed:
    - Use the modifier const before the parameter type
    - The parameter becomes a constant parameter
    - const used in the function declaration and definition

Chapter 10

Defining Classes

What Is a Class?

- A class is a data type that is defined by a user to represent an object.
- Most of the data types we have used are built-in types, such as
  - `int, char, float, double, long, char*`
- We can define our own data types using
  - typedefs -> typedef type TypeName;
  - structs
  - classes

Class Definitions

- A class definition includes
  - Properties and functions that apply to the entire class [class variables, class methods]
  - Properties that are common to every member. [instance variables]
  - Functions that are available to every member. [methods or member functions]

- We will start by defining structures as a first step toward defining classes

Overview

10.1 Structures

10.2 Classes

10.3 Abstract Data Types

10.4 Introduction to Inheritance

10.1

Structures

Structures

- A structure can be viewed as an object
  - Used when multiple values are needed to describe an object. Examples?
  - Doesn't need to contain member functions (The structures used here have no member functions)
  - Contains multiple values of possibly different types
    - The multiple values are logically related as a single item
    - Example: A bank Certificate of Deposit (CD)
      has the following values:
      a balance
      an interest rate
      a term (months to maturity)

The CD Definition

- The Certificate of Deposit structure can be defined as

```
struct CDAccount
{
      double balance;
      double interest_rate;
      int term;  //months to maturity
};
```

Remember this semicolon!

- Keyword `struct` begins a structure definition
- `CDAccount` is the structure tag
- Member names are identifiers declared in the braces

Using the Structure

- **Structure definition is generally placed outside any function definition in the global space.**
  - This makes the structure type available to all code that follows the structure definition
- To declare two variables of type CDAccount:

```
CDAccount  my_account, your_account;
```

  - My_account and your_account contain distinct member variables balance, interest_rate, and term

The Structure Value

- The Value of a Structure
  - Consists of the values of the member variables of the structure

- The value of an object of type CDAccount
  - Consists of the values of the member variables

```
        balance
        interest_rate
        term
```

Specifying Member Variables

- Member variables are specific to the structure variable in which they are declared

  - Syntax to specify a member variable:

    ```
    Structure_Variable_Name.Member_Variable_Name
    ```

  - Given the declaration:

    ```
    CDAccount  my_account, your_account;
    ```

    - Use the dot operator to specify a member variable

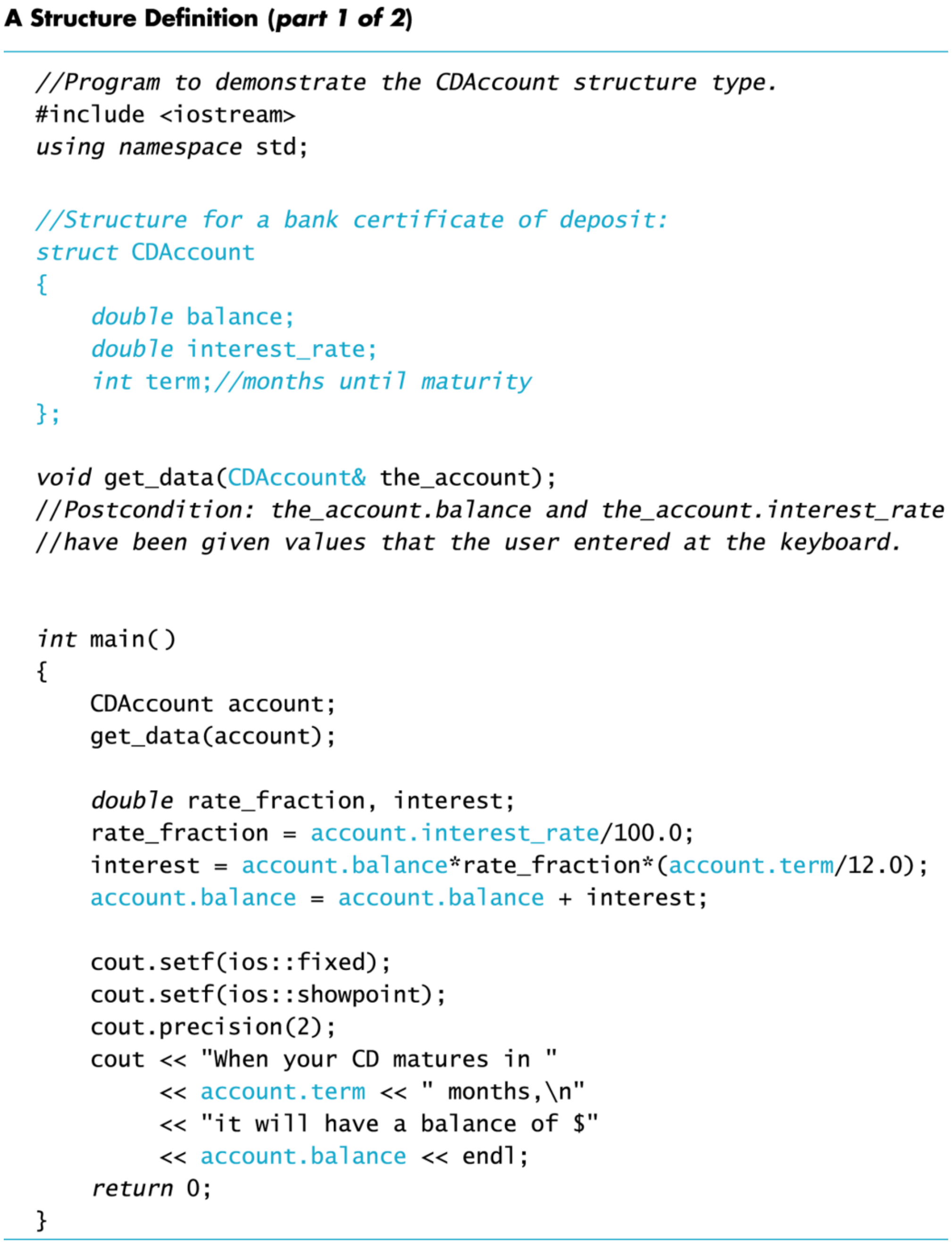
      ```
      my_account.balance
      my_account.interest_rate
      my_account.term
      ```

Using Member Variables

- Member variables can be used just as any other variable of the same type
  - `my_account.balance = 1000;`
    `your_account.balance = 2500;`
    - Notice that `my_account.balance` and `your_account.balance` are different variables!
  - `my_account.balance = my_account.balance + interest;`

Display 10.1 (1)

Display 10.1 (2)

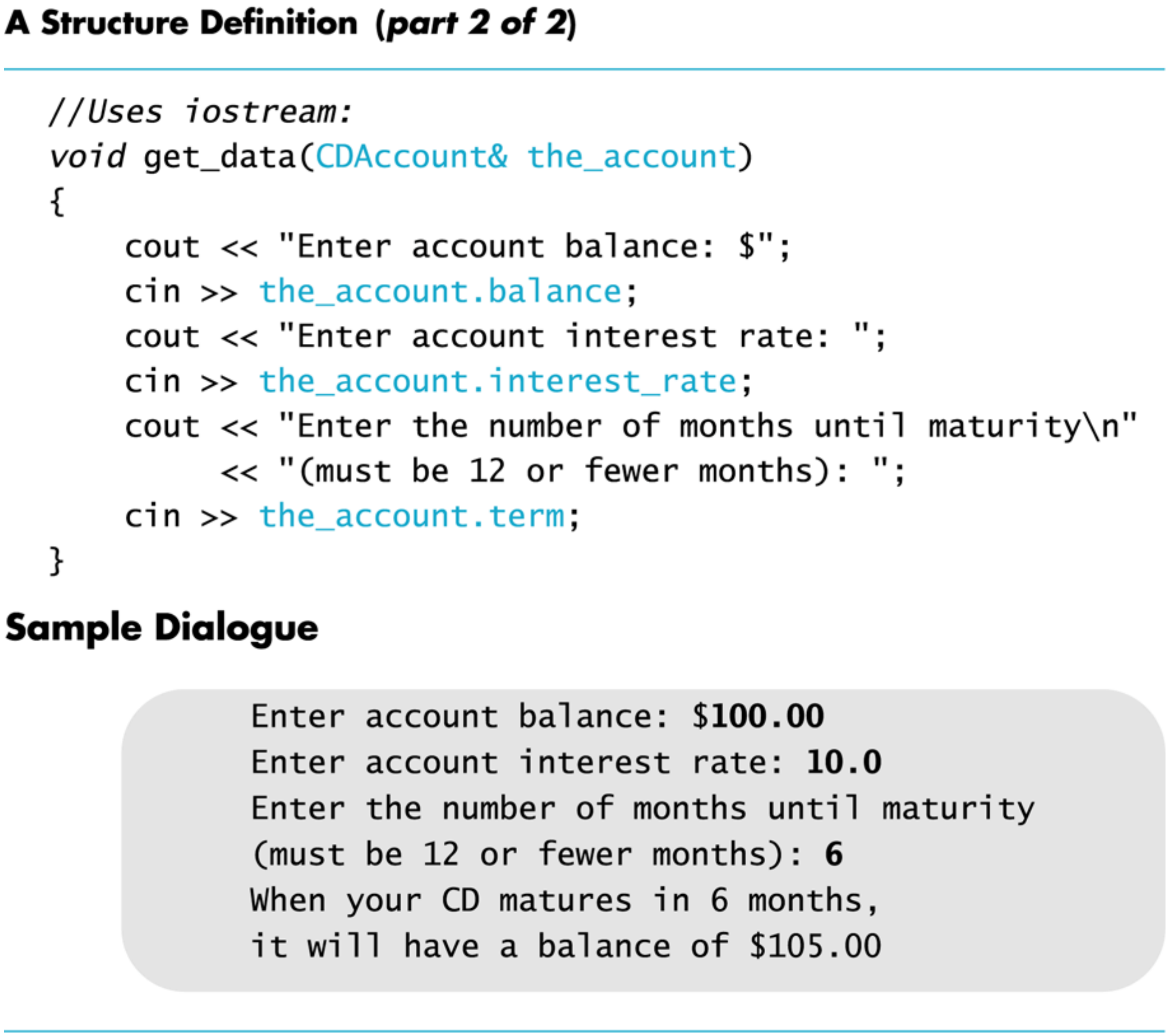


A Structure Definition (*part 1 of 2*)

```
//Program to demonstrate the CDAccount structure type.
#include <iostream>
using namespace std;

//Structure for a bank certificate of deposit:
struct CDAccount
{
    double balance;
    double interest_rate;
    int term;//months until maturity
};

void get_data(CDAccount& the_account);
//Postcondition: the_account.balance and the_account.interest_rate
//have been given values that the user entered at the keyboard.


int main()
{
    CDAccount account;
    get_data(account);

    double rate_fraction, interest;
    rate_fraction = account.interest_rate/100.0;
    interest = account.balance*rate_fraction*(account.term/12.0);
    account.balance = account.balance + interest;

    cout.setf(ios::fixed);
    cout.setf(ios::showpoint);
    cout.precision(2);
    cout << "When your CD matures in "
         << account.term << " months,\n"
         << "it will have a balance of $"
         << account.balance << endl;
    return 0;
}
```

Display 10.1 (1/2)

Display 10.1 (2/2)



A Structure Definition ***(part 2 of 2)***

```
//Uses iostream:
void get_data(CDAccount& the_account)
{
    cout << "Enter account balance: $";
    cin >> the_account.balance;
    cout << "Enter account interest rate: ";
    cin >> the_account.interest_rate;
    cout << "Enter the number of months until maturity\n"
         << "(must be 12 or fewer months): ";
    cin >> the_account.term;
}
```

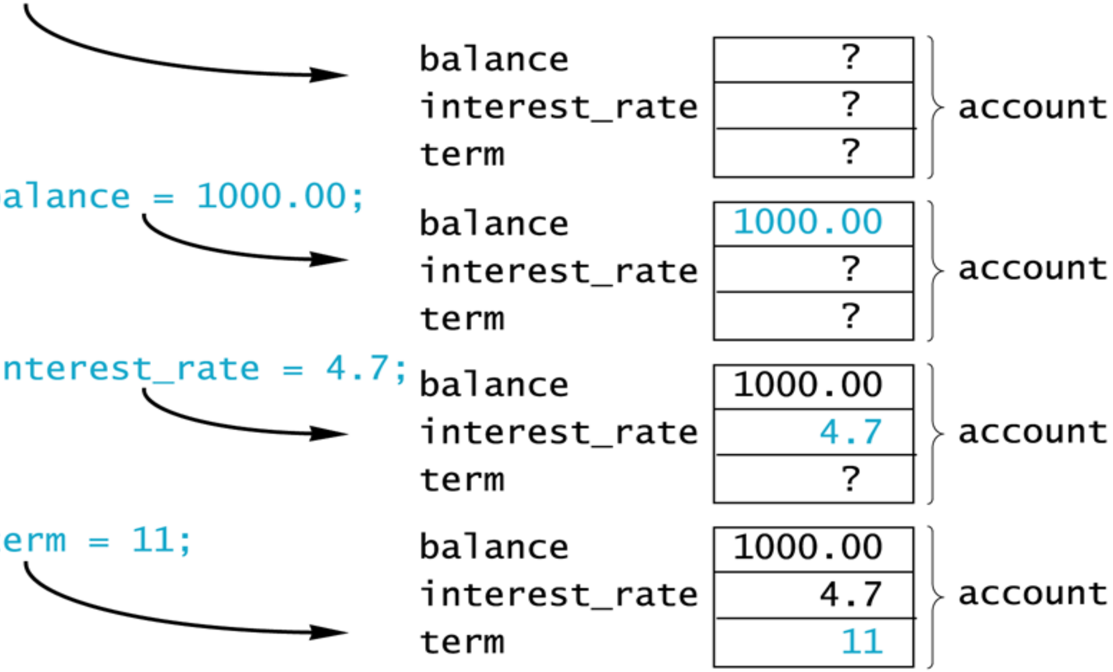
Sample Dialogue

```
Enter account balance: $100.00
Enter account interest rate: 10.0
Enter the number of months until maturity
(must be 12 or fewer months): 6
When your CD matures in 6 months,
it will have a balance of $105.00
```

Display 10.2

Member Values

```
struct CDAccount
{
    double balance;
    double interest_rate;
    int term;//months until maturity
};
int main()
{
  CDAccount account;
    ...

  account.balance = 1000.00;

  account.interest_rate = 4.7;

  account.term = 11;
```



Duplicate Names

- Member variable names duplicated between structure types are not a problem.

```
struct FertilizerStock
{
   double quantity;
   double nitrogen_content;
};

FertilizerStock  super_grow;
```

```
struct CropYield
{
   int quantity;
   double size;
};

CropYield  apples;
```

- `super_grow.quantity` and `apples.quantity` are different variables stored in different locations

Structures as Arguments

- Structures can be arguments in function calls
  - The formal parameter can be call-by-value
  - The formal parameter can be call-by-reference
- Example:

```
void get_data(CDAccount& the_account);
```

  - Uses the structure type CDAccount we saw earlier as the type for a call-by-reference parameter

Assignment and Structures

- The assignment operator can be used to assign values to structure types
- Using the CDAccount structure again:

```
CDAccount my_account, your_account;
my_account.balance = 1000.00;
my_account.interest_rate = 5.1;
my_account.term = 12;
your_account = my_account;
```

  - Assigns all member variables in your_account the corresponding values in my_account

Structures as Return Types

- Structures can be the type of a value returned by a function.
- Example:

```
CDAccount shrink_wrap(double the_balance,
                      double the_rate,
                      int the_term)
{
    CDAccount temp;
    temp.balance = the_balance;
    temp.interest_rate = the_rate;
    temp.term = the_term;
    return temp;
}
```

Using Function shrink_wrap

- shrink_wrap builds a complete structure value in temp, which is returned by the function
- We can use shrink_wrap to give a variable of type CDAccount a value in this way:

```
CDAccount  new_account;
new_account = shrink_wrap(1000.00, 5.1, 11);
```

The above assignment operator copies the whole structure content (given by the return statement) into new_account.

Hierarchical Structures

- Structures can contain member variables that are also structures

```
struct Date
{
   int month;
   int day;
   int year;
};
```

```
struct PersonInfo
{
   double height;
   int weight;
   Date birthday;
};
```

- `struct PersonInfo` contains a `Date` structure

Using PersonInfo

- A variable of type `PersonInfo` is declared by

```
PersonInfo person1;
```

- To display the birth year of `person1`, first access the `birthday` member of `person1`

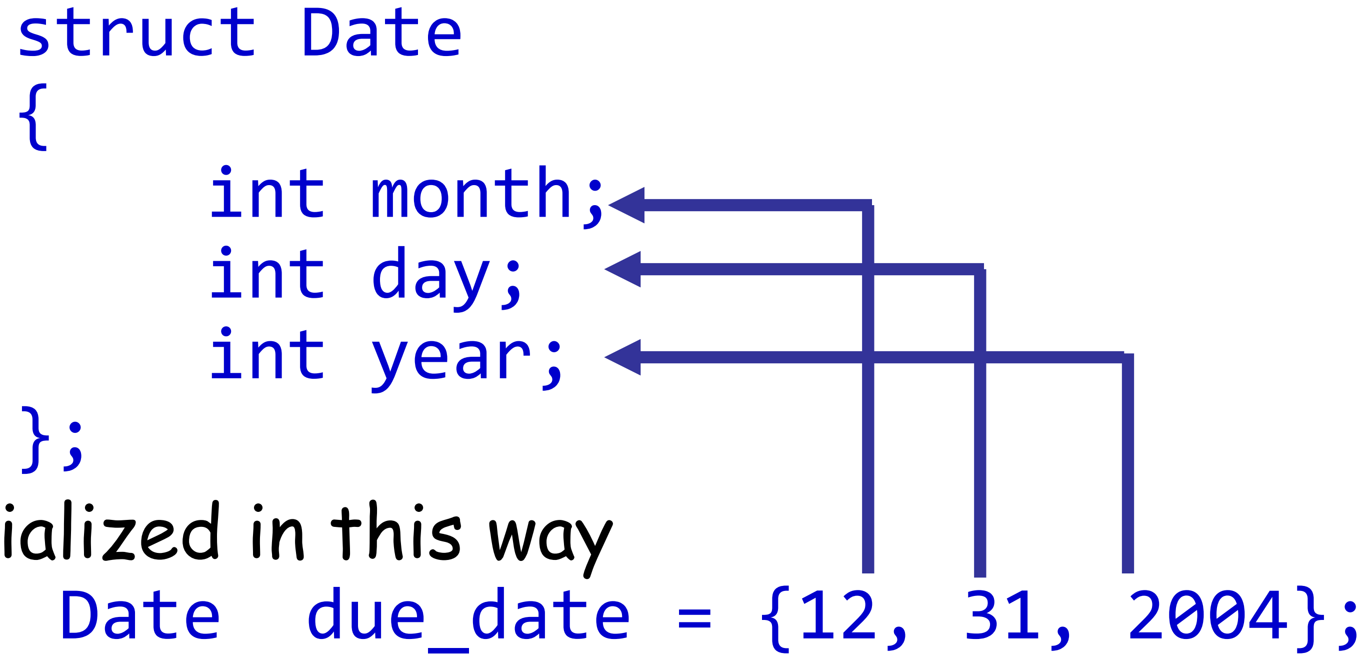
```
cout <<  person1.birthday…
```

- But we want the year, so we now specify the year member of the birthday member

```
cout << person1.birthday.year;
```

Initializing Classes

- A structure can be initialized when declared
- Example:



```
struct Date
{
        int month;
        int day;
        int year;
};
```

Can be initialized in this way

```
  Date  due_date = {12, 31, 2004};
```

Compare with array initialization

Section 10.1 Exercise

- Can you

  - Write a definition for a structure type for records consisting of a person's wage rate, accrued vacation (in whole days), and status (hourly or salaried). Represent the status as one of the two character values 'H' and 'S'. Call the type EmployeeRecord.

Structure EmployeeRecord

- Define a structure called EmployeeRecord that has the following fields:
  - A wage rate – to hold a monetary value
  - Accrued vacation – in whole days
  - A wage status for hourly vs salaried

- What types do these fields have?

Struct EmployeeRecord

- Now we can definition the EmployeeRecord

```
struct EmployeeRecord {
      double wage_rate;   // hourly or annual rate
      int    accrued_vacation; // in whole days
      char wage_status; // ‘H’ = hourly; ‘S’=annual
};
```

STRUCTS (AND CLASSES) AS PARAMETERS

Passing structs as arguments

- Pass-by-values copies the struct member by member onto the stack into the parameters.

  - waste of time and space.

- Pass-by-reference copies only the address of the object onto the stack but the argument can be changed.

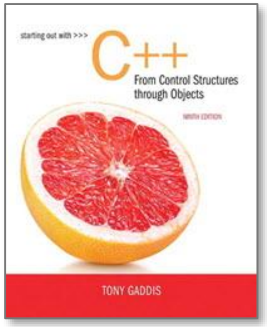
  - dangerous unless change is expected.

- We want an efficient way to pass objects (structs and classes) as arguments.

Struct as Function Arguments

- Using value parameter for structure can slow down a program, waste space with object copy
- Using a reference parameter will speed up program, but function may change data in structure
- Using a `const` reference parameter allows read-only access to reference parameter, does not waste space, speed

```
void displayDate(const Date& date) {
     cout << date.month << ‘/’ <<
           date.day << ‘/’ << date.year;
}
```



11.9

Pointers to Structures

Pointers to Structures

- A structure variable has an address
- Pointers to structures are variables that can hold the address of a structure:

```
Student *stuPtr;
```

- Can use `&` operator to assign address:

```
stuPtr = & stu1;
```

- Structure pointer can be a function parameter

Accessing Structure Members via Pointer Variables

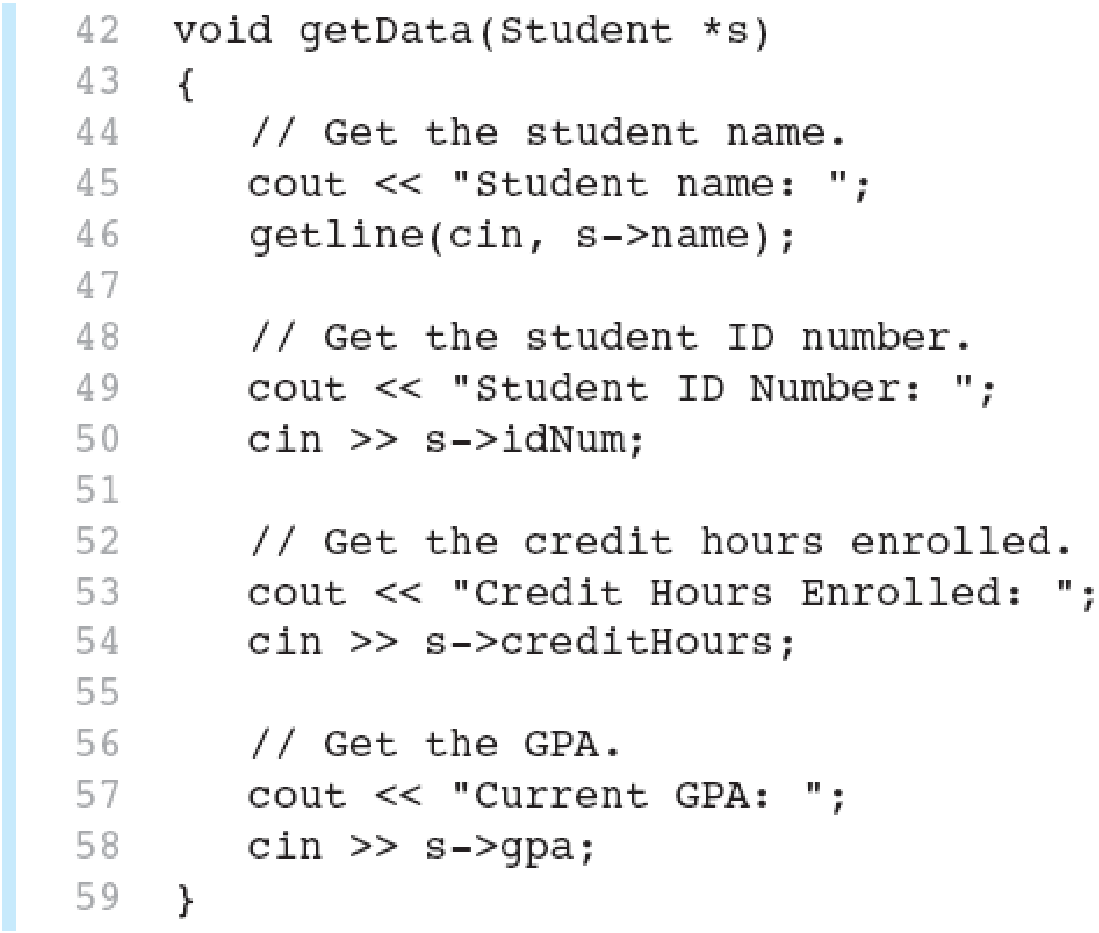
- Must use `()` to dereference pointer variable, not field within structure:

```
cout << (*stuPtr).studentID;
```

- Can use structure pointer operator to eliminate `()` and use clearer notation:

```
cout << stuPtr->studentID;
```

From Program 11-8



```
void getData(Student *s)
{
   // Get the student name.
   cout << "Student name: ";
   getline(cin, s->name);

   // Get the student ID number.
   cout << "Student ID Number: ";
   cin >> s->idNum;

   // Get the credit hours enrolled.
   cout << "Credit Hours Enrolled: ";
   cin >> s->creditHours;

   // Get the GPA.
   cout << "Current GPA: ";
   cin >> s->gpa;
}
```

10.2

Classes

Classes

- A class is a data type whose variables are called objects.
  - The definition of a class includes
    - Description of the kinds of values of the member variables
    - Description of the member functions
  - A class description is like a structure definition except that members are **private** by default…

A Class Example

- To create a new type named DayOfYear as a class definition
  - Decide on the values to represent
  - This example's values are dates such as July 4 using an integer for the number of the month
    - Member variable month is an int (Jan = 1, Feb = 2, etc.)
    - Member variable day is an int
  - Decide on the member functions needed
  - We use just one member function named output

Class DayOfYear Definition

```
class DayOfYear
{
      public:
              void output( );
               int month;
               int day;
};
```

Member Function **Declaration**

Struct vs Class – the true difference!

- Structs have public data members and functions by default
  - Access struct members directly using the ‘.’ (dot) operator:

    ```
    cout << bday.month << “/” << bday.year;
    ```

- Classes have private data members and functions by default.
  - Access class members using member functions called with the ‘.’ (dot) operator

    ```
    bday.output();
    ```

Public or Private Members

- The keyword `public` identifies the members of a class that can be accessed from outside the class
  - Members that follow the keyword `public` are public members of the class (can be accessed by anyone)
- The keyword `private` identifies the members of a class that can be accessed only by member functions of the class (can only be accessed by class)
  - Members that follow the keyword `private` are private members of the class

Defining a Member Function

- Member functions are **declared** in the class declaration
- Member function **definitions** identify the class in which the function is a member

```
void DayOfYear::output()
{
      cout << “month = “ << month
            <<  “,  day = “ << day
            << endl;
}
```

Member Function Definition

- Member function definition syntax:

```
Returned_Type Class_Name::Function_Name(Parameter_List)
{
            Function Body Statements
}
```

  - Example:

```
void DayOfYear::output( )
{
   cout << “month = “ << month
    << “, day = “ << day << endl;
}
```

The '::' Operator

- '::' is the **scope resolution** operator
  - Tells the class a member function is a member of
  - `void DayOfYear::output( )` indicates that function `output` is a member of the `DayOfYear` class
  - The class name that precedes '::' is a type qualifier

'::' and '.'

:: used with **classes** to identify a member

```
void DayOfYear::output( )
{
 // function body
}
```

. used with **variables (or objects)** to identify a member

```
DayOfYear birthday;
birthday.output( );
```

Calling Member Functions

- Calling the DayOfYear member function output is done in this way:

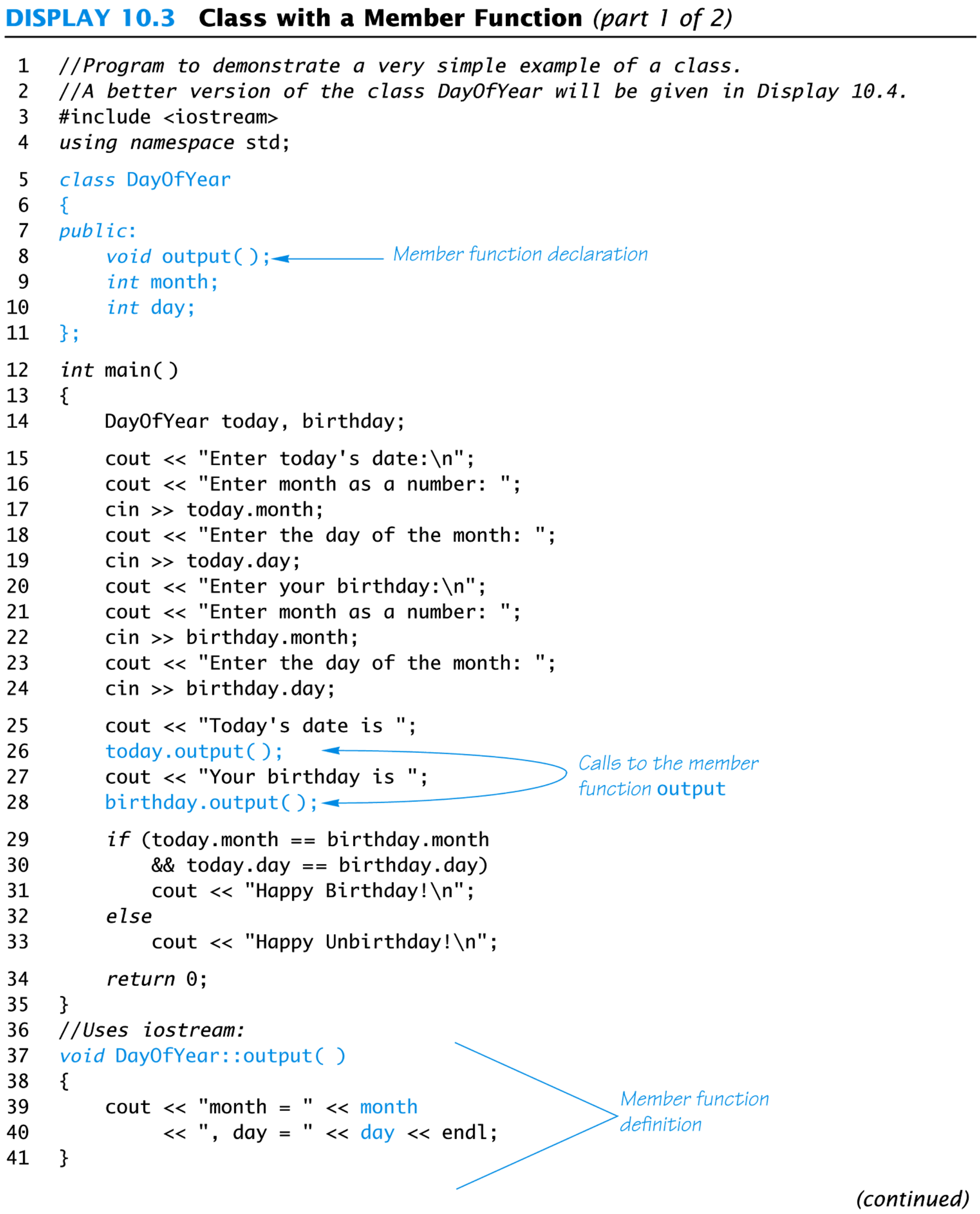
```
DayOfYear today, birthday;
today.output( );
birthday.output( );
```

  - Note that today and birthday have their own versions of the month and day variables for use by the output function

Display 10.3 (1)

Display 10.3 (2)

Display 10.3 (1/2)



DISPLAY 10.3 Class with a Member Function *(part 1 of 2)*

```
//Program to demonstrate a very simple example of a class.
//A better version of the class DayOfYear will be given in Display 10.4.
#include <iostream>
using namespace std;

class DayOfYear
{
public:
    void output( );
    int month;
    int day;
};

int main( )
{
    DayOfYear today, birthday;

    cout << "Enter today's date:\n";
    cout << "Enter month as a number: ";
    cin >> today.month;
    cout << "Enter the day of the month: ";
    cin >> today.day;
    cout << "Enter your birthday:\n";
    cout << "Enter month as a number: ";
    cin >> birthday.month;
    cout << "Enter the day of the month: ";
    cin >> birthday.day;

    cout << "Today's date is ";
    today.output( );
    cout << "Your birthday is ";
    birthday.output( );

    if (today.month == birthday.month
        && today.day == birthday.day)
        cout << "Happy Birthday!\n";
    else
        cout << "Happy Unbirthday!\n";

    return 0;
}
//Uses iostream:
void DayOfYear::output( )
{
    cout << "month = " << month
         << ", day = " << day << endl;
}
```

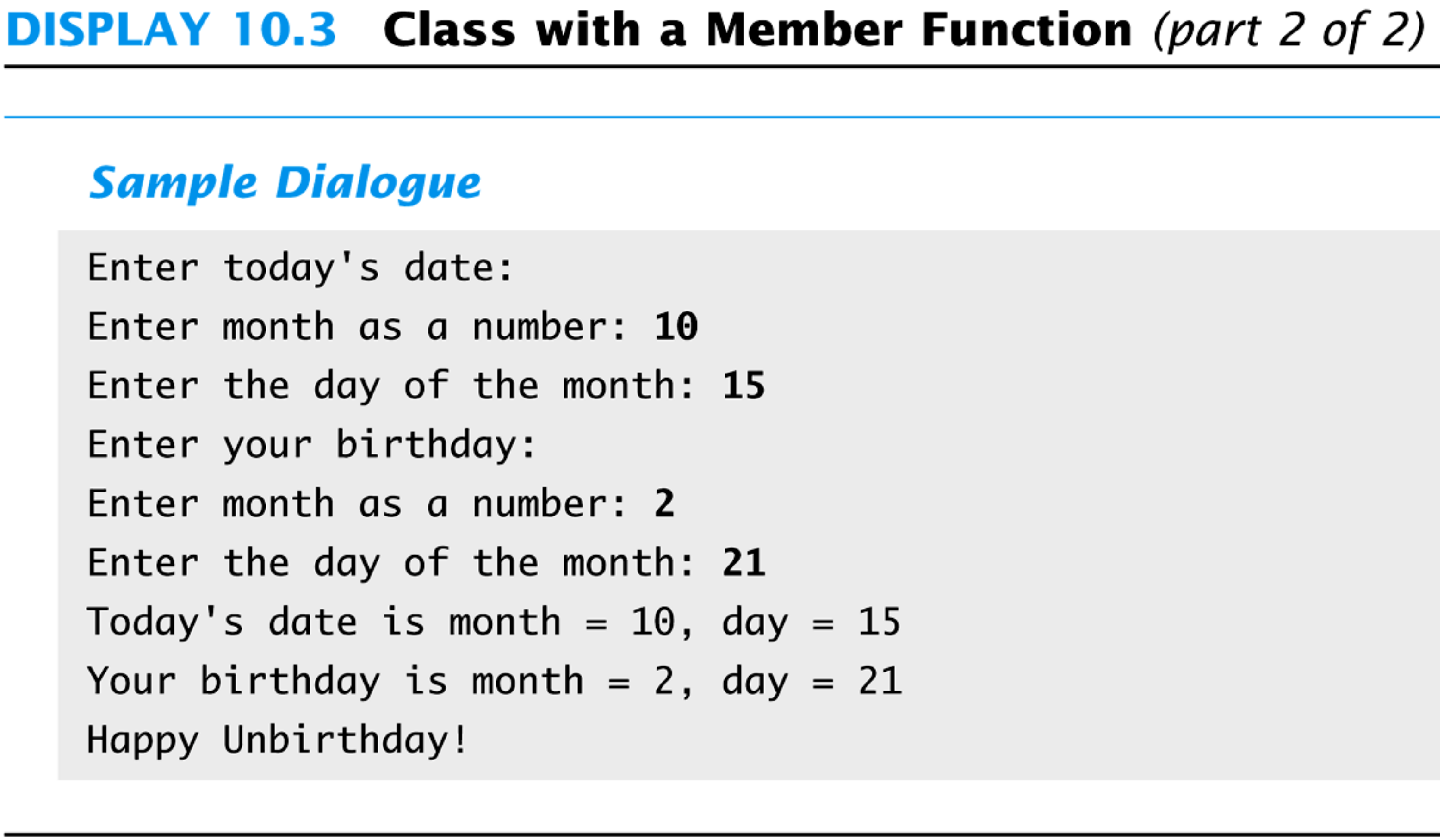
Member function declaration

Calls to the member function output

Member function definition

(continued)

Display 10.3 (2/2)



DISPLAY 10.3 **Class with a Member Function** *(part 2 of 2)*

Sample Dialogue

```
Enter today's date:
Enter month as a number: 10
Enter the day of the month: 15
Enter your birthday:
Enter month as a number: 2
Enter the day of the month: 21
Today's date is month = 10, day = 15
Your birthday is month = 2, day = 21
Happy Unbirthday!
```

Ideal Class Definitions

- Changing the implementation of `DayOfYear` requires changes to the program that uses `DayOfYear`
- An ideal class definition of `DayOfYear` could be changed without requiring changes to the program that uses `DayOfYear`

Problems With DayOfYear

- Changing how the month is stored in the class DayOfYear requires changes to the main program
- If we decide to store the month as four characters (JAN, FEB, etc.) instead of an int
  - `cin >> today.month` will no longer work because we now have three character variables to read
  - `if(today.month == birthday.month)` will no longer work to compare months
  - The member function "output" no longer works

Fixing DayOfYear

- To fix DayOfYear
  - We need to add member functions to use when changing or accessing the member variables
    - If the program (that uses DayOfYear) **never directly references** the member variables of DayOfYear, changing how the variables are stored will not require changing the program
  - We need to be sure that the program does not ever directly reference the member variables

Public Or Private?

- C++ helps us restrict the program from **directly referencing** member variables
  - **Private** members of a class can only be referenced within the definitions of member functions
    - If the program (other than through member functions) tries to access a private member, the compiler gives an error message
  - **Private** members can be variables or functions

Private Variables

- Private variables cannot be accessed directly by the program
  - Changing their values requires the use of public member functions of the class
  - To set the private month and day variables in a new DayOfYear class use a member function such as

```
void DayOfYear::set(int new_month, int new_day)
{
  month = new_month;
  day = new_day;
}
```

Public or Private Members

- The keyword `private` identifies the members of a class that can be accessed only by member functions of the class
  - Members that follow the keyword `private` are private members of the class
- The keyword `public` identifies the members of a class that can be accessed from outside the class
  - Members that follow the keyword `public` are public members of the class

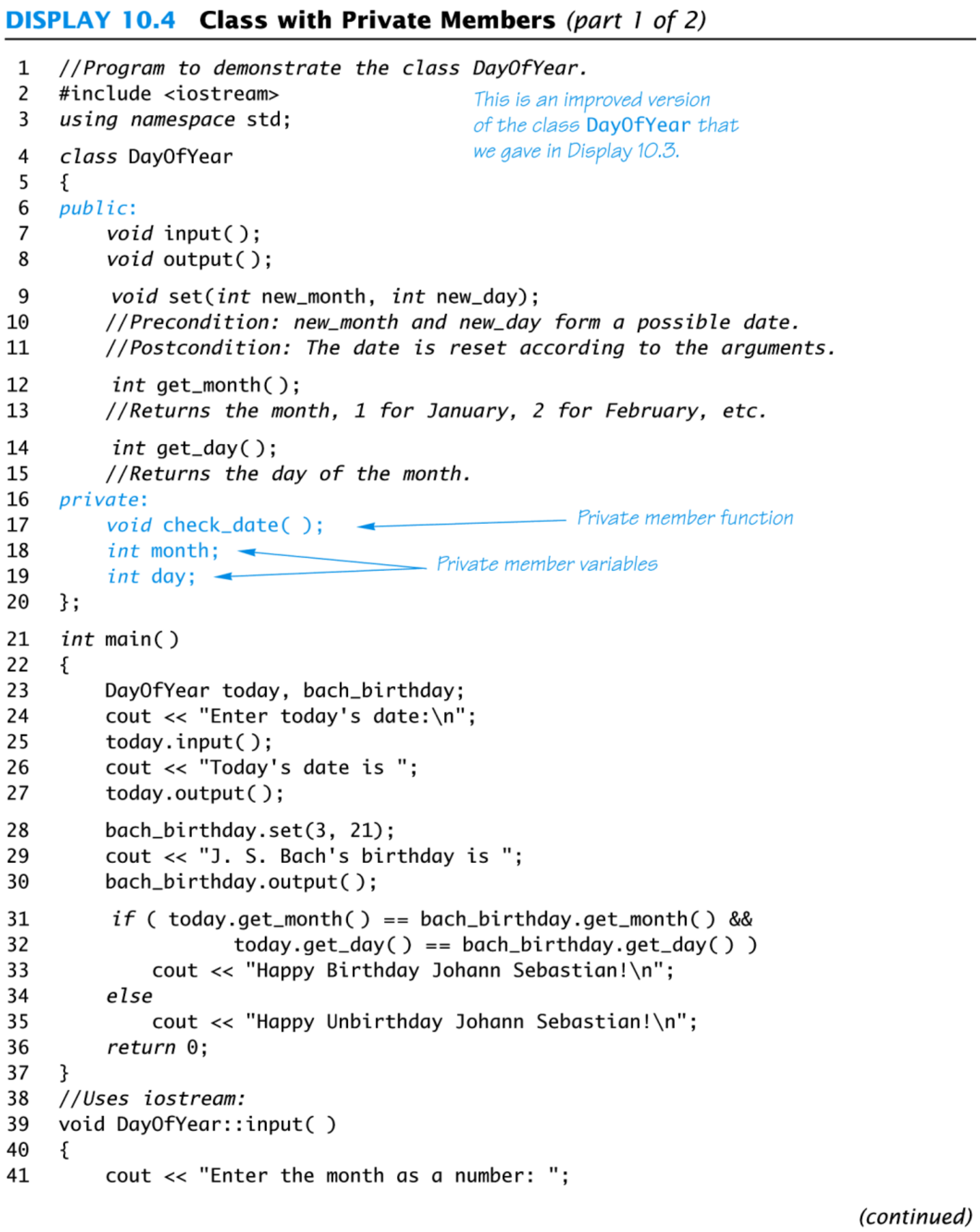
A New DayOfYear

- The new DayOfYear class demonstrated in Display 10.4…
  - All member variables are private
  - Uses member functions to do all manipulation of the **private** member variables
    - **Private** member variables and member function definitions can be changed without changes to the program that uses DayOfYear

Display 10.4 (1)

Display 10.4 (2)

Display 10.4 (1/2)



DISPLAY 10.4 Class with Private Members *(part 1 of 2)*

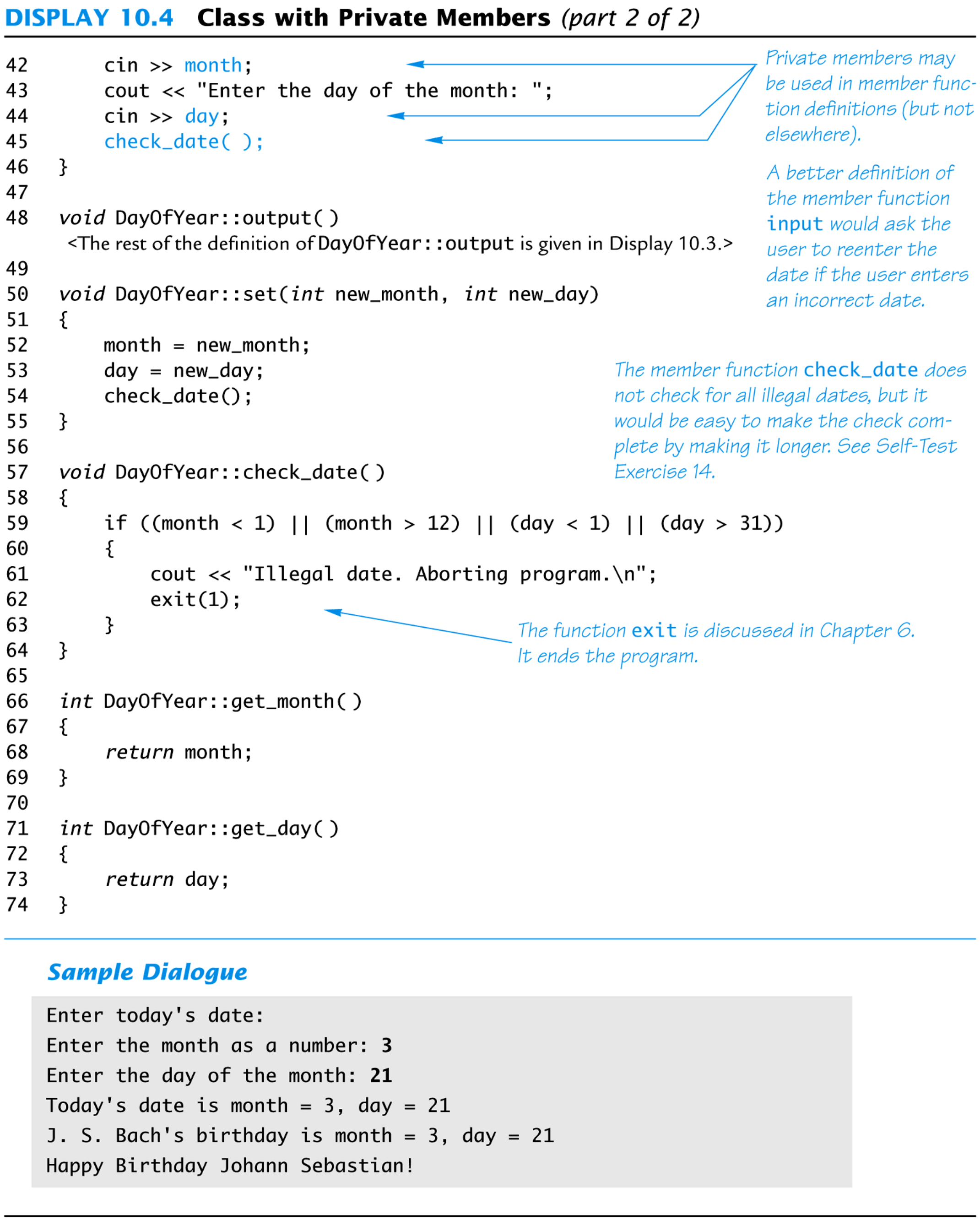
```
//Program to demonstrate the class DayOfYear.
#include <iostream>
using namespace std;
class DayOfYear
{
public:
    void input( );
    void output( );
    void set(int new_month, int new_day);
    //Precondition: new_month and new_day form a possible date.
    //Postcondition: The date is reset according to the arguments.
    int get_month( );
    //Returns the month, 1 for January, 2 for February, etc.
    int get_day( );
    //Returns the day of the month.
private:
    void check_date( );
    int month;
    int day;
};
int main( )
{
    DayOfYear today, bach_birthday;
    cout << "Enter today's date:\n";
    today.input( );
    cout << "Today's date is ";
    today.output( );
    bach_birthday.set(3, 21);
    cout << "J. S. Bach's birthday is ";
    bach_birthday.output( );
    if ( today.get_month( ) == bach_birthday.get_month( ) &&
              today.get_day( ) == bach_birthday.get_day( ) )
        cout << "Happy Birthday Johann Sebastian!\n";
    else
        cout << "Happy Unbirthday Johann Sebastian!\n";
    return 0;
}
//Uses iostream:
void DayOfYear::input( )
{
    cout << "Enter the month as a number: ";
```

This is an improved version of the class DayOfYear that we gave in Display 10.3.

Private member function

Private member variables

(continued)



DISPLAY 10.4 Class with Private Members *(part 2 of 2)*

```
    cin >> month;
    cout << "Enter the day of the month: ";
    cin >> day;
    check_date( );
}

void DayOfYear::output( )
 <The rest of the definition of DayOfYear::output is given in Display 10.3.>

void DayOfYear::set(int new_month, int new_day)
{
    month = new_month;
    day = new_day;
    check_date();
}

void DayOfYear::check_date( )
{
    if ((month < 1) || (month > 12) || (day < 1) || (day > 31))
    {
        cout << "Illegal date. Aborting program.\n";
        exit(1);
    }
}

int DayOfYear::get_month( )
{
    return month;
}

int DayOfYear::get_day( )
{
    return day;
}
```

Private members may be used in member function definitions (but not elsewhere).

A better definition of the member function **input** *would ask the user to reenter the date if the user enters an incorrect date.*

The member function **check_date** *does not check for all illegal dates, but it would be easy to make the check complete by making it longer. See Self-Test Exercise 14.*

The function **exit** *is discussed in Chapter 6. It ends the program.*

Sample Dialogue

```
Enter today's date:
Enter the month as a number: 3
Enter the day of the month: 21
Today's date is month = 3, day = 21
J. S. Bach's birthday is month = 3, day = 21
Happy Birthday Johann Sebastian!
```

Display 10.4 (2/2)

Ideal Class Definitions

- Changing the implementation of DayOfYear requires changes to the program that uses DayOfYear
- An ideal class definition of DayOfYear could be changed without requiring changes to the program that uses DayOfYear
- Member functions for
  - set(int month, int day), input(), output()
  - get_month() and get_day()

Using Private Variables

- **It is normal to make all member variables private**
- Private variables require member functions to perform all changing and retrieving of values
  - **Accessor** functions allow you to obtain the values of member variables
    - Example: get_day in class DayOfYear
  - **Mutator** functions allow you to change the values of member variables
    - Example: set in class DayOfYear
- *Another term for Accessor is* **getter** and Mutator is **setter**.

Even more Ideal

- Implement a member function that tells if one DayOfYear object is equal to another.
- Remember that to call a member function requires an object and the '.' operator.
    - What is the return type?
    - What is the parameter type?
    - What is the code?

bool isEqual(DayOfYear doy)

```
bool DayOfYear::isEqual(DayOfYear doy)
{
   if (month == doy.get_month() &&
       day == doy.get_day())
       return true;
   return false;
}
```

General Class Definitions

- The syntax for a class definition is

```
class Class_Name
 {
   public:
        Member_Specification_1
        Member_Specification_2
        …
        Member_Specification_3
   private:
        Member_Specification_n+1
        Member_Specification_n+2
        …
 };
```

The Assignment Operator

- Objects and structures can be assigned values with the assignment operator (=)
  - Example:

```
DayOfYear  due_date, tomorrow;

tomorrow.set(11, 19);

due_date = tomorrow;
```

Structs and classes syntax so far…

Structs

```
struct NewStruct {
        type variable1;
        …
};
void passReference(NewStruct& ns);
NewStruct shrink_wrap(var1,…);

NewStruct aStructVar;
NewStruct bStructVar;
// Assignment is supported
aStructVar = shrink_wrap(var1,…);
bStructVar = aStructVar;
```

Classes

```
class NewClass {
public:
        type set(type var1,…);
private:
        type privMemberFunc();
        type variable1;
};
void passReference(NewClass& nc);
// Assignment is supported
NewClass aClassVar, bClassVar;
aClassVar.set(var1, …);
bClassVar = aClassVar;
```

Declaring an Object

- Once a class is defined, an object of the class is declared just as variables of any other type
  - Example:

To create two objects of type Bicycle:

```
class Bicycle
 {
       // class definition lines
 };

 Bicycle my_bike,  your_bike;
```

Bicycle as a struct or class

```
struct Bicycle { // all members public
     int wheel_height;
     int num_wheels;
     int num_gears;
}; // separate functions to set values
Bicycle initBicycle(int wht, int nw, int ng)
{
   Bicycle tempBike;
    tempBike.wheel_height = wht;
    tempBike.num_wheels = nw;
    tempBike.num_gears = ng;
    return tempBike;
}
// myBike and yourBike are distinct.
Bicycle myBike = initBicycle(15,2,10);
Bicycle yourBike = myBike;
```

```
class Bicycle { // public methods
   public:
       void set(int wht, int nw, int ng);
       …
   private: // private members, functions
      int wheel_height;
      int num_wheels;
      int num_gears;
}; // member functions access fields directly
void Bicycle::set(int wht, int nw, int ng) {
      wheel_height = wht;
      num_wheels = nw;
      num_gears = ng;
}
Bicycle myBike.set(15, 2, 10);
```

Encapsulation

- Encapsulation is
  - Combining a number of items, such as variables and functions, into a single package such as an object of a class
  - Keeps the data or properties together with the functions that operate on them.
- Why is encapsulation desirable?
  - Reusable, Maintainable.

Structs and classes

- When do we use structs? When do we use classes?
  - Mostly, we use classes but…
  - Structs are useful when code has to be accessible to both c and c++.
  - Structs are useful to describe an object that is internal to a class or if direct access to data members is required
  - But even then, it's probably better to just make it a class.

Consider a rectangle class

- What is needed to describe a rectangle?
- What functions can we do with a rectangle?
- Consider using a rectangle in math class?

Class Rectangle

```
class Rectangle {
   public:
      int getHeight() const;
      int getWidth() const;
      int calcPerimeter() const;
      int calcArea() const;
      void set(int h, int w);
      void addTo(int hDim, int wDim);
      void draw();
  private:
      int height;
      int width;
};
```

Public Members

Private Members

Access Specifiers

- Used to control access to members of the class

- `public:` can be accessed by functions outside of the class

- `private:` can only be called by or accessed by functions that are members of the class

More on Access Specifiers

- Can be listed in any order in a class
- Can appear multiple times in a class
- If not specified, the default is `private`

Constructors

- Member function that is automatically called when an object is created
- Purpose is to construct an object
- Constructor function name is class name
- Has no return type

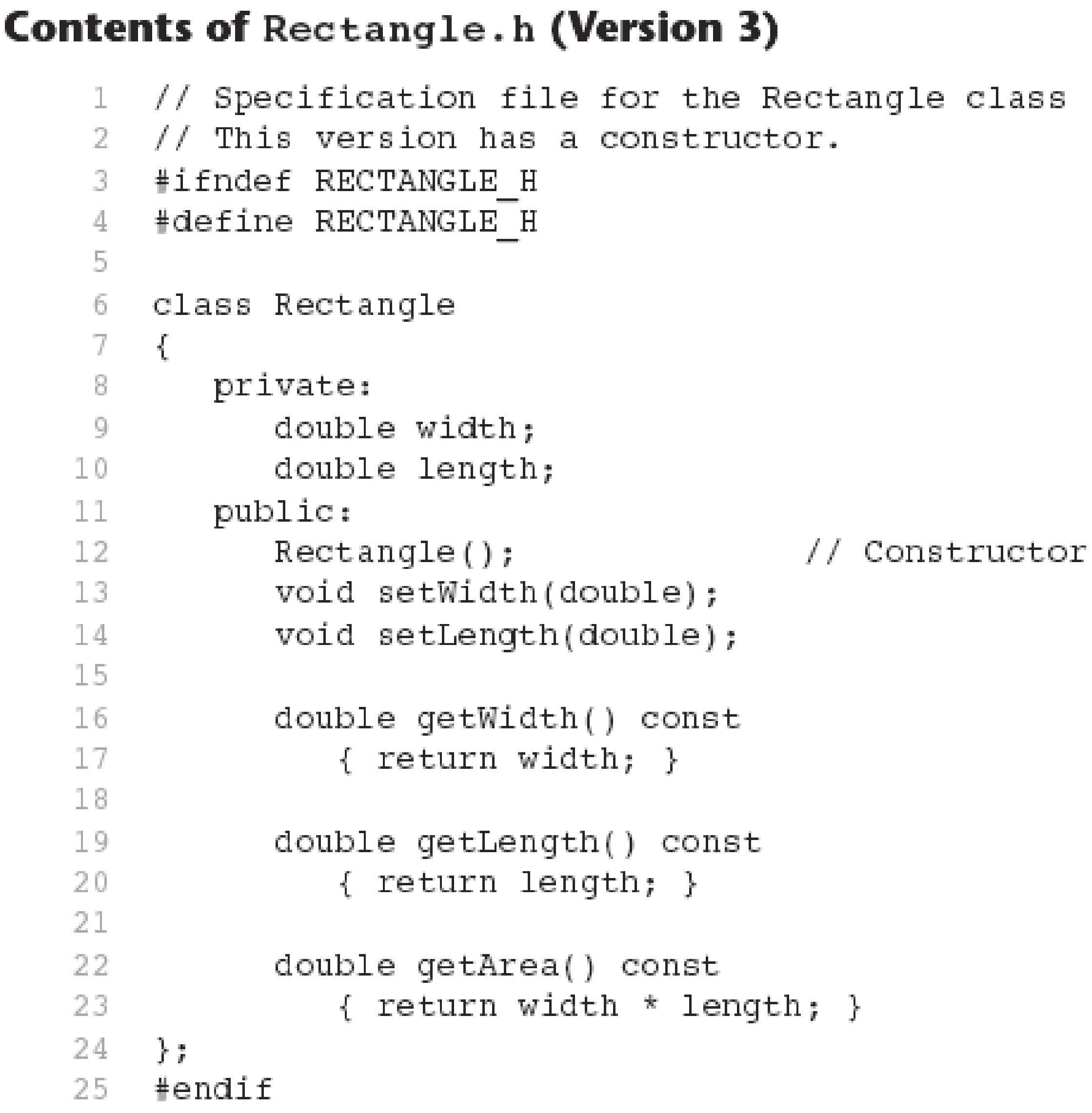
Default Constructors

- A default constructor is a constructor that takes no arguments.

- If you write a class with no constructor at all, C++ will write a default constructor for you, one that does nothing.

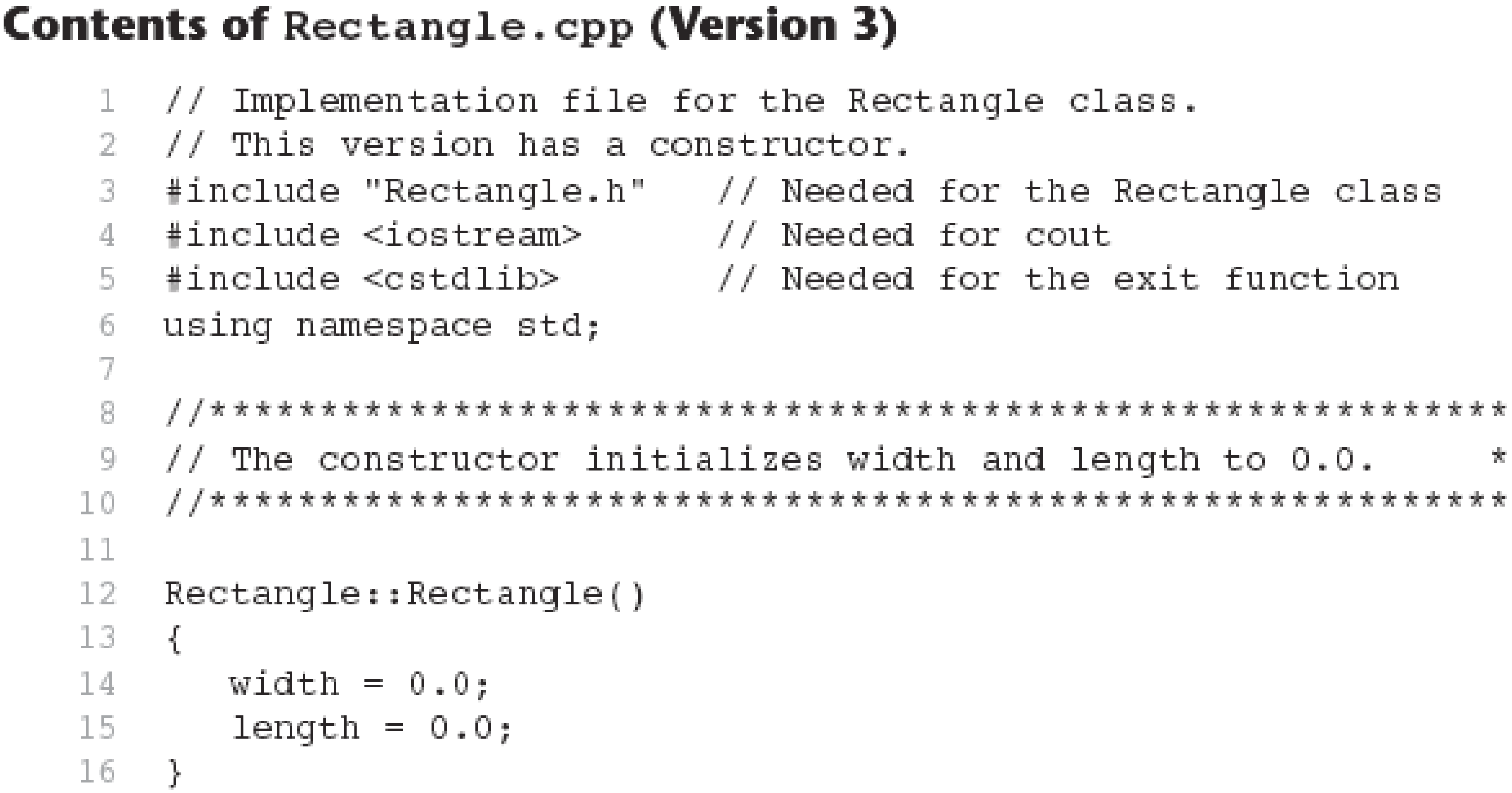
- A simple instantiation of a class (with no arguments) calls the default constructor:

```
Rectangle r;
```



Contents of Rectangle.h (Version 3)

```
// Specification file for the Rectangle class
// This version has a constructor.
#ifndef RECTANGLE_H
#define RECTANGLE_H

class Rectangle
{
   private:
      double width;
      double length;
   public:
      Rectangle();               // Constructor
      void setWidth(double);
      void setLength(double);

      double getWidth() const
         { return width; }

      double getLength() const
         { return length; }

      double getArea() const
         { return width * length; }
};
#endif
```

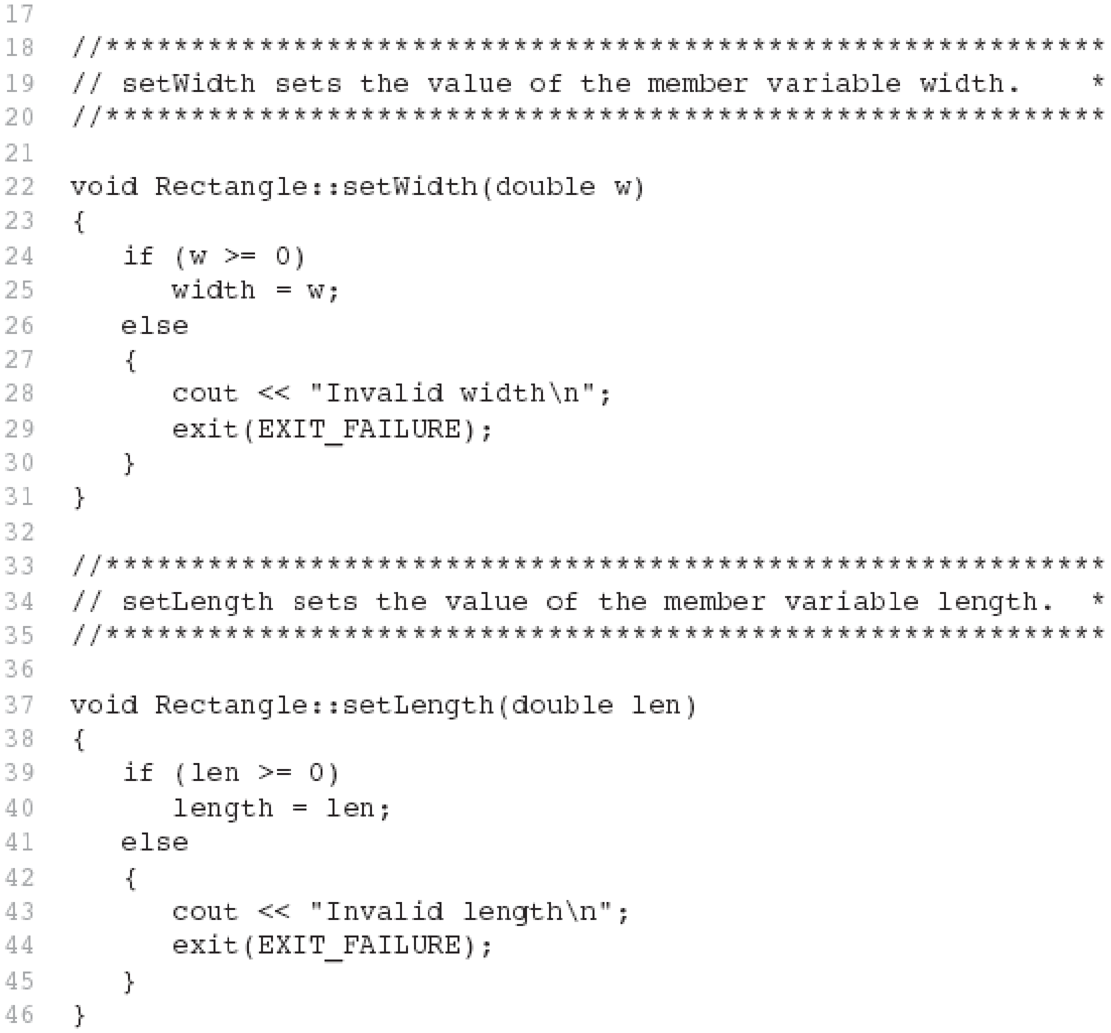


Contents of Rectangle.cpp (Version 3)

```
// Implementation file for the Rectangle class.
// This version has a constructor.
#include "Rectangle.h"   // Needed for the Rectangle class
#include <iostream>      // Needed for cout
#include <cstdlib>       // Needed for the exit function
using namespace std;

//***********************************************************
// The constructor initializes width and length to 0.0.     *
//***********************************************************

Rectangle::Rectangle()
{
   width = 0.0;
   length = 0.0;
}
```

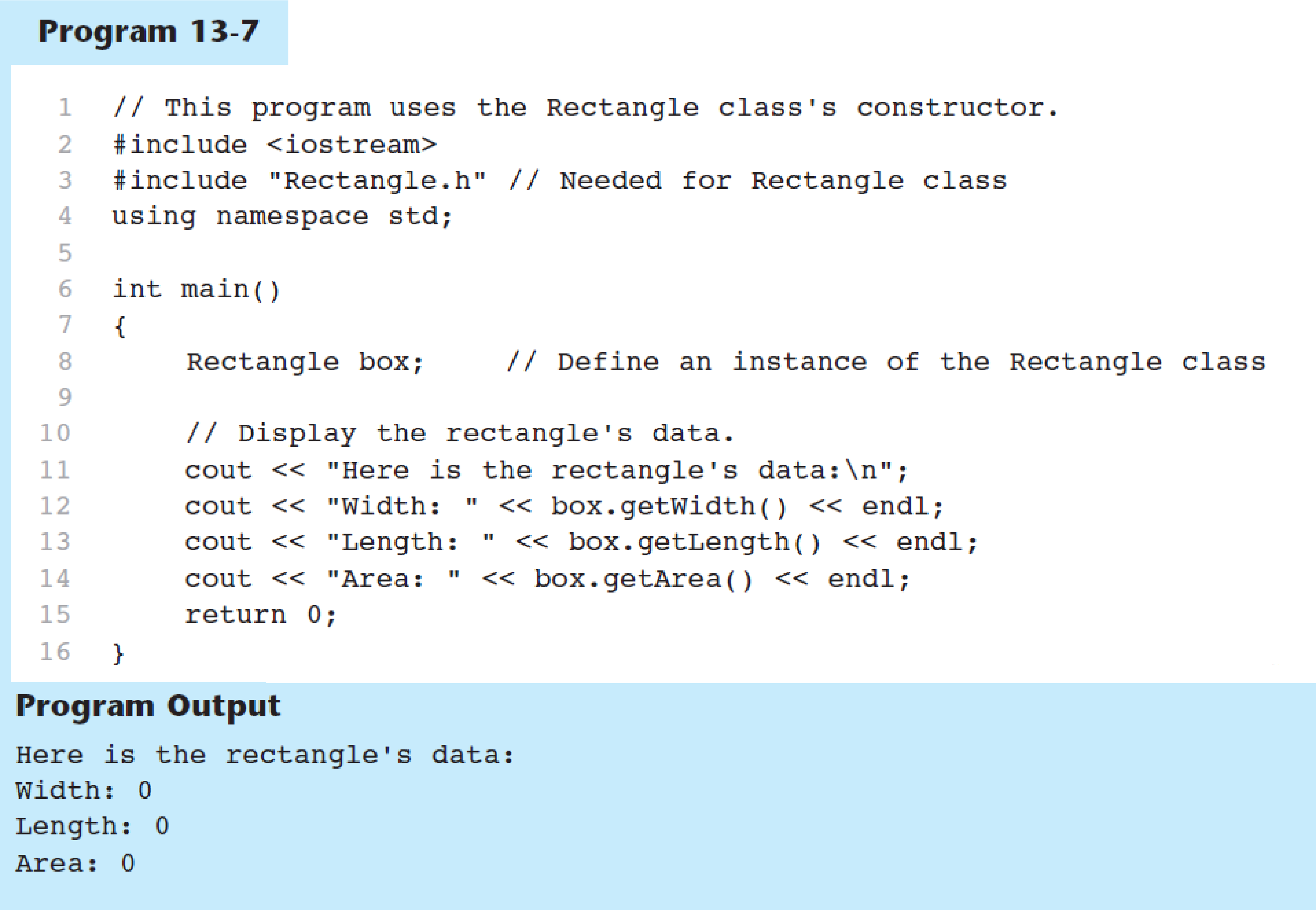
Continues...

Contents of `Rectangle.ccp` Version3



```
//***********************************************************
// setWidth sets the value of the member variable width.    *
//***********************************************************

void Rectangle::setWidth(double w)
{
   if (w >= 0)
      width = w;
   else
   {
      cout << "Invalid width\n";
      exit(EXIT_FAILURE);
   }
}

//***********************************************************
// setLength sets the value of the member variable length.  *
//***********************************************************

void Rectangle::setLength(double len)
{
   if (len >= 0)
      length = len;
   else
   {
      cout << "Invalid length\n";
      exit(EXIT_FAILURE);
   }
}
```

(continued)



Program 13-7

```cpp
// This program uses the Rectangle class's constructor.
#include <iostream>
#include "Rectangle.h" // Needed for Rectangle class
using namespace std;

int main()
{
    Rectangle box;     // Define an instance of the Rectangle class

    // Display the rectangle's data.
    cout << "Here is the rectangle's data:\n";
    cout << "Width: " << box.getWidth() << endl;
    cout << "Length: " << box.getLength() << endl;
    cout << "Area: " << box.getArea() << endl;
    return 0;
}
```

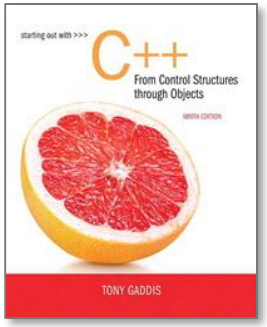
Program Output

```
Here is the rectangle's data:
Width: 0
Length: 0
Area: 0
```

Default Constructors

- If your program does not provide any constructor for a class defined by you, C++ generates a default one for you that does nothing.
- If your program does provide some constructor (maybe only one), but no default constructor, C++ does NOT generate a default one.

See the demo program…



13.8

Passing Arguments to Constructors

Passing Arguments to Constructors

- To create a constructor that takes arguments:
  - indicate parameters in prototype:

    ```
    Rectangle(double, double);
    ```

  - Use parameters in the definition:

    ```
    Rectangle::Rectangle(double w, double
    len)
    {
        width = w;
        length = len;
    }
    ```

Passing Arguments to Constructors

- You can pass arguments to the constructor when you create an object:

```
Rectangle r(10, 5);
```

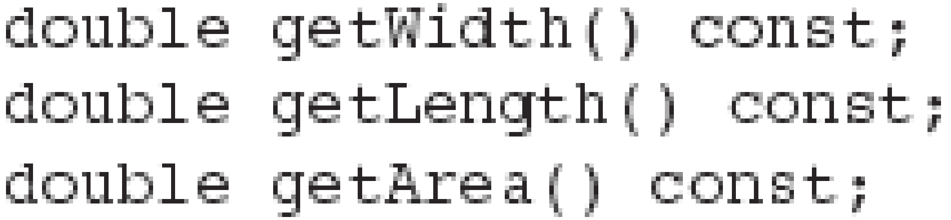
Classes with No Default Constructor

- When all of a class's constructors require arguments, then the class has NO default constructor.

- When this is the case, you must pass the required arguments to the constructor when creating an object.

Using `const` With Member Functions

- `const` appearing after the parentheses in a member function declaration specifies that the function will not change any data in the calling object.



```
double getWidth() const;
double getLength() const;
double getArea() const;
```

Defining a Member Function

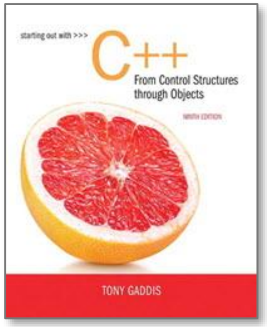
- When defining a member function:
  - Put prototype in class declaration
  - Define function using class name and scope resolution operator `(::)`

```
int Rectangle::setWidth(double w)
{
    width = w;
}
```

Remember Accessors and Mutators

- Mutator: a member function that stores a value in a private member variable, or changes its value in some way

- Accessor: function that retrieves a value from a private member variable. Accessors do not change an object's data, so they should be marked `const`.



13.3

Defining an Instance of a Class

Defining an Instance of a Class

- An object is an instance of a class
- Defined like structure variables:

```
Rectangle r;
```

- Access members using dot operator:

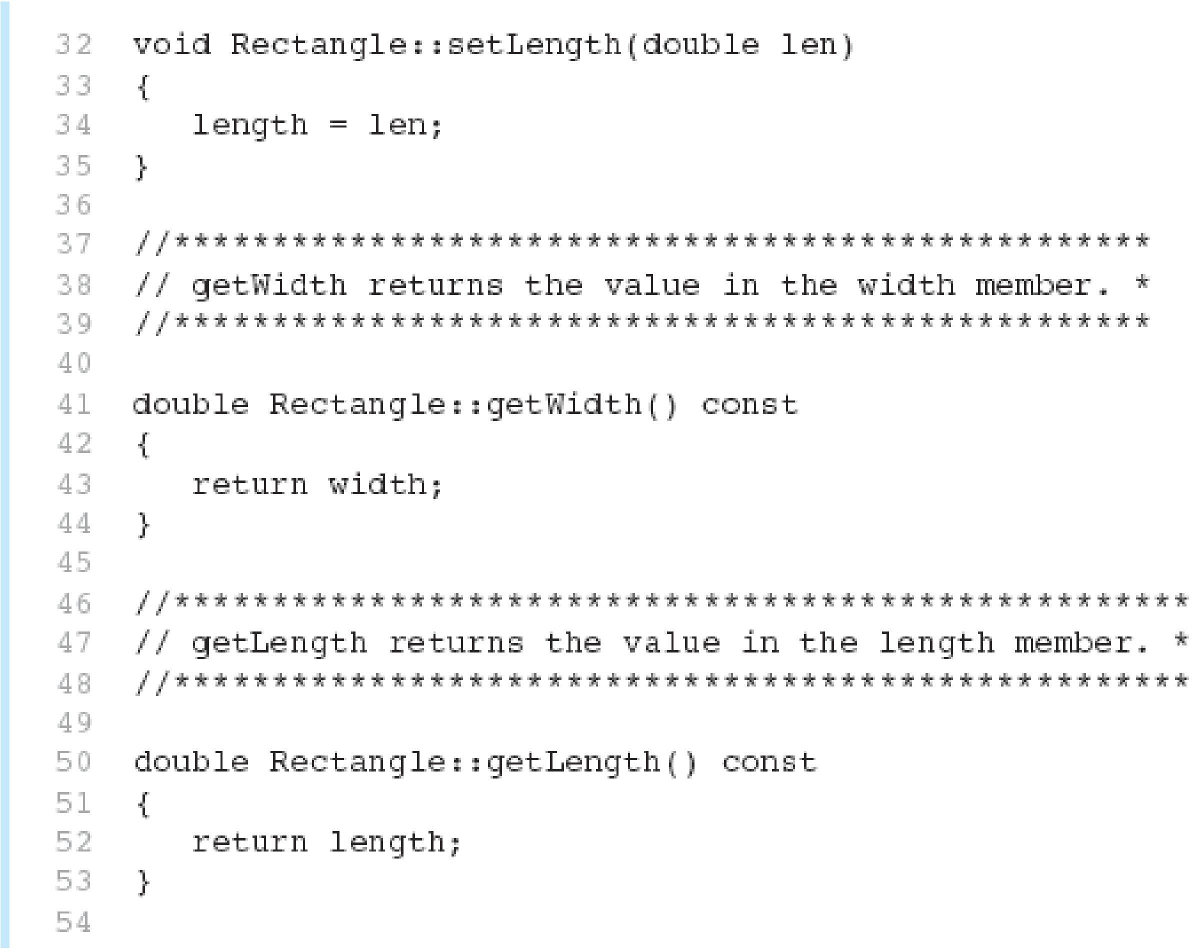
```
r.setWidth(5.2);
cout << r.getWidth();
```

- Compiler error if attempt to access `private` member using dot operator

Program 13-1

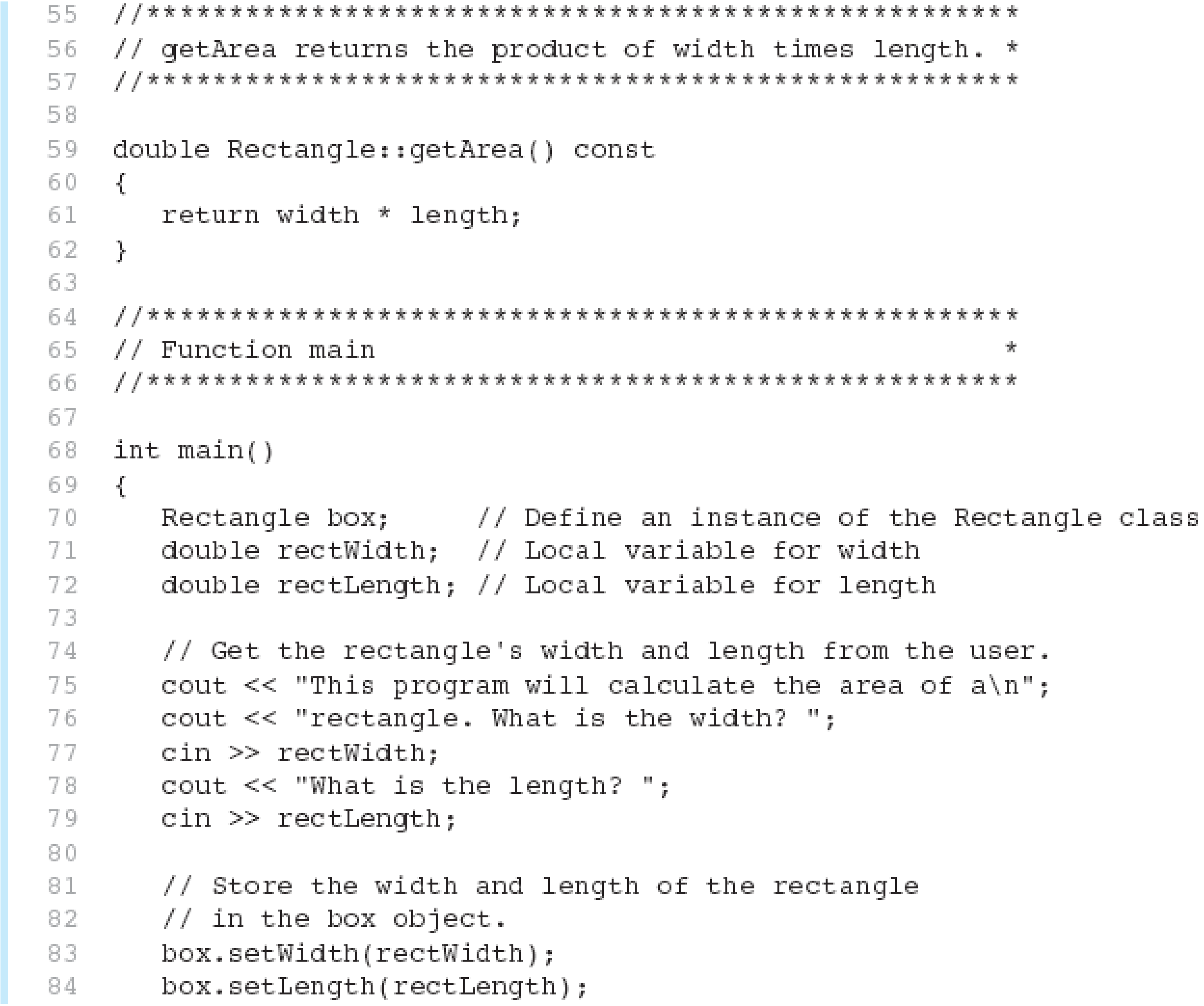
```
// This program demonstrates a simple class.
#include <iostream>
using namespace std;

// Rectangle class declaration.
class Rectangle
{
   private:
      double width;
      double length;
   public:
      void setWidth(double);
      void setLength(double);
      double getWidth() const;
      double getLength() const;
      double getArea() const;
};

//**************************************************
// setWidth assigns a value to the width member.   *
//**************************************************

void Rectangle::setWidth(double w)
{
   width = w;
}

//**************************************************
// setLength assigns a value to the length member. *
//**************************************************

```

Program 13-1 (Continued)



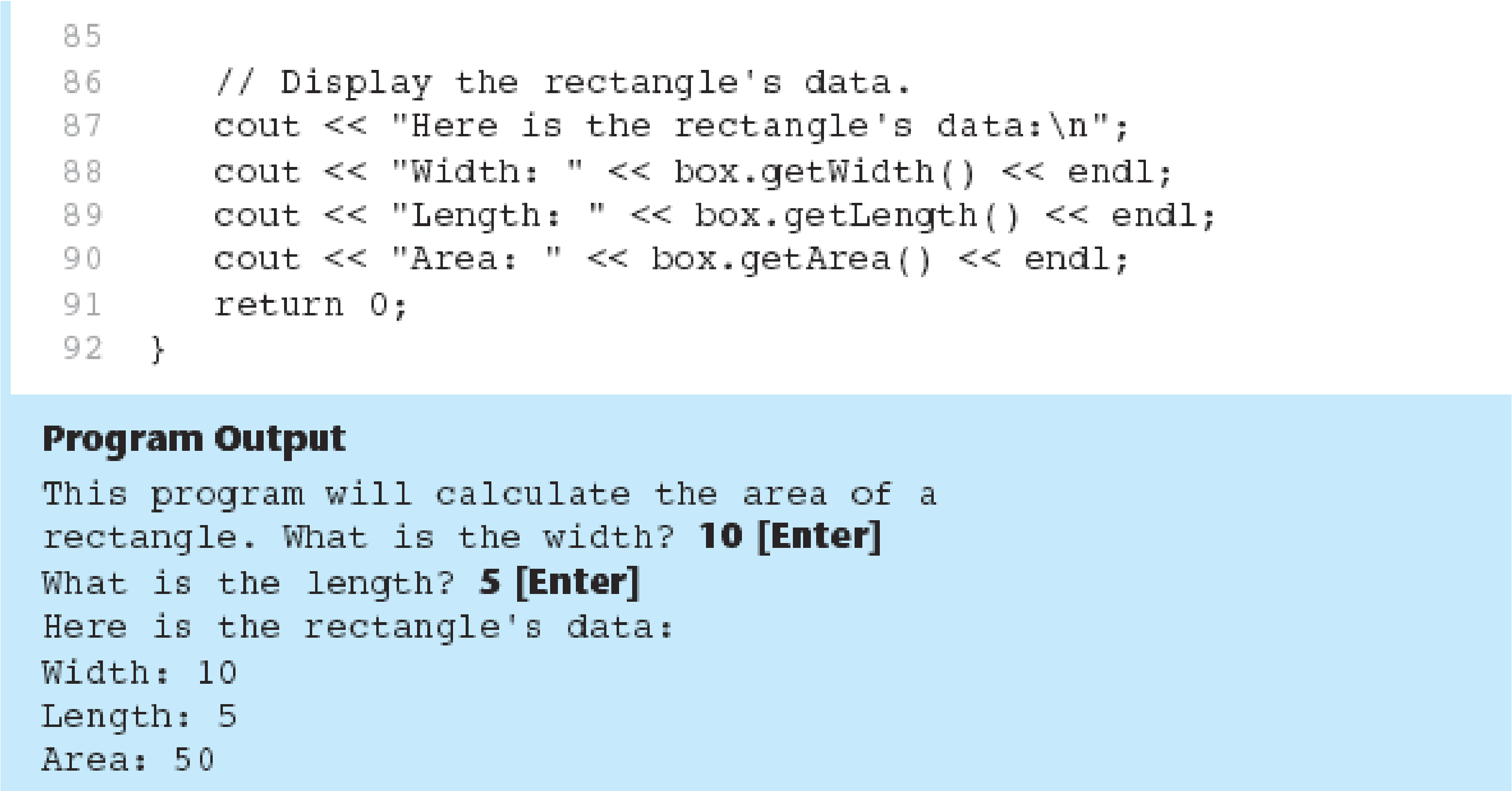
```
void Rectangle::setLength(double len)
{
   length = len;
}

//**************************************************
// getWidth returns the value in the width member. *
//**************************************************

double Rectangle::getWidth() const
{
   return width;
}

//****************************************************
// getLength returns the value in the length member. *
//****************************************************

double Rectangle::getLength() const
{
   return length;
}

```

Program 13-1 (Continued)



```
//*****************************************************
// getArea returns the product of width times length. *
//*****************************************************

double Rectangle::getArea() const
{
   return width * length;
}

//*****************************************************
// Function main                                      *
//*****************************************************

int main()
{
   Rectangle box;     // Define an instance of the Rectangle class
   double rectWidth;  // Local variable for width
   double rectLength; // Local variable for length

   // Get the rectangle's width and length from the user.
   cout << "This program will calculate the area of a\n";
   cout << "rectangle. What is the width? ";
   cin >> rectWidth;
   cout << "What is the length? ";
   cin >> rectLength;

   // Store the width and length of the rectangle
   // in the box object.
   box.setWidth(rectWidth);
   box.setLength(rectLength);
```

Program 13-1 (Continued)



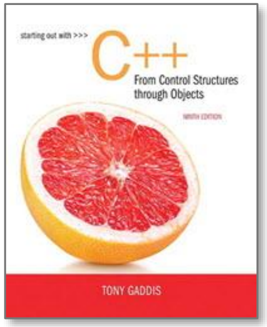
```

    // Display the rectangle's data.
    cout << "Here is the rectangle's data:\n";
    cout << "Width: " << box.getWidth() << endl;
    cout << "Length: " << box.getLength() << endl;
    cout << "Area: " << box.getArea() << endl;
    return 0;
}
```

Program Output

```
This program will calculate the area of a
rectangle. What is the width? 10 [Enter]
What is the length? 5 [Enter]
Here is the rectangle's data:
Width: 10
Length: 5
Area: 50
```

Avoiding Stale Data

- Some data is the result of a calculation.
- In the `Rectangle` class the area of a rectangle is calculated.
  - length x width
- If we were to use an `area` variable here in the `Rectangle` class, its value would be dependent on the length and the width.
- If we change `length` or `width` without updating `area`, then `area` would become *stale*.
- To avoid stale data, it is best to calculate the value of that data within a member function rather than store it in a variable.



13.10

Overloading Constructors

Overloading Constructors

- A class can have more than one constructor

- Overloaded constructors in a class must have different parameter lists:

```
Rectangle();
Rectangle(double);
Rectangle(double, double);
```

More About Default Constructors

- If all of a constructor's parameters have default arguments, then it is a default constructor. Done in declaration. For example:

```
 Rectangle(double w= 0, double l= 0);
```

- Creating an object and passing no arguments will cause this constructor to execute:

```
Rectangle r;
```

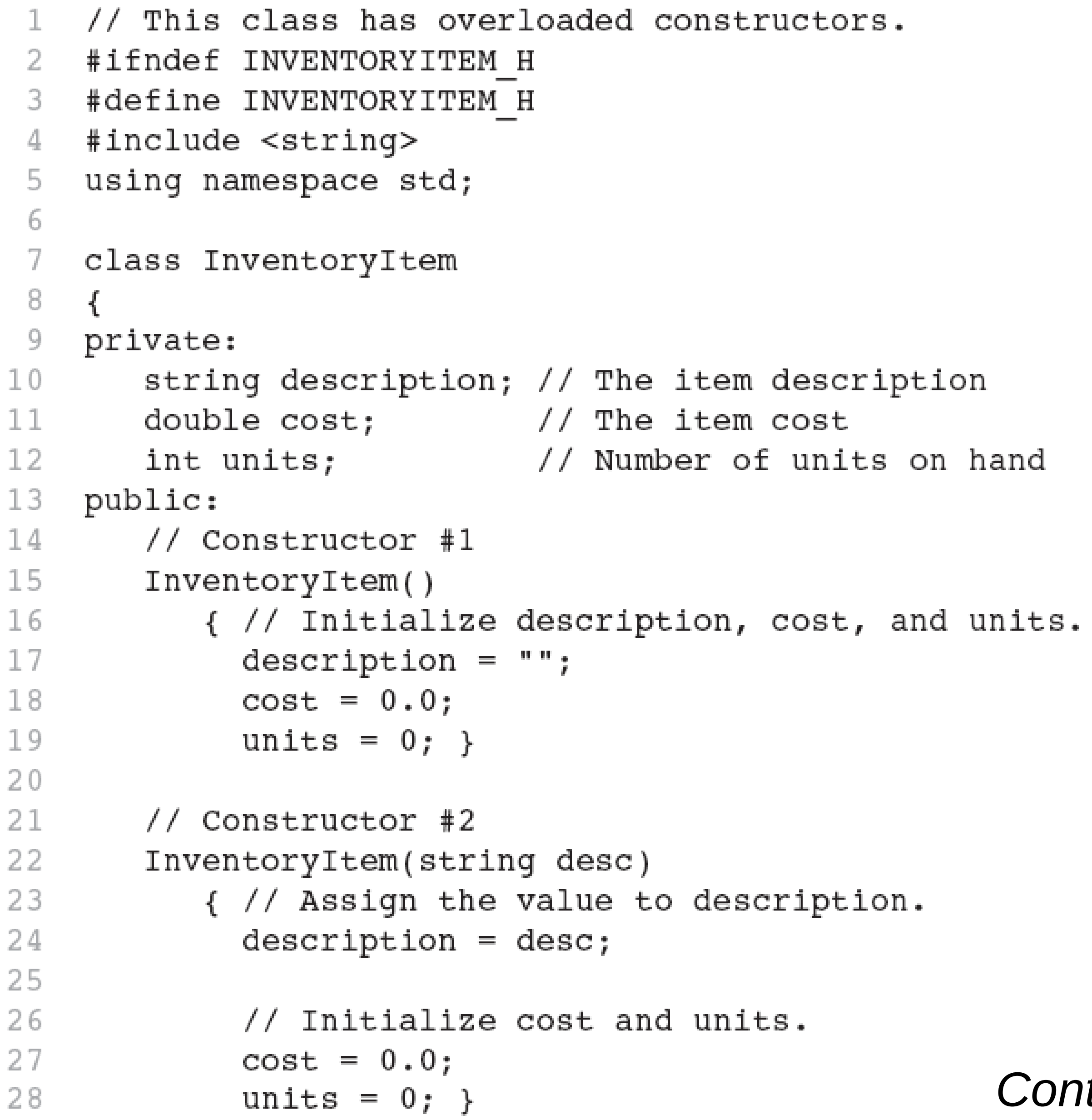
Initialization Section

- C++ constructors have a special section that can be used to initialize data members:

```
 Rectangle::Rectangle(int w, int l) :
width(w), length(l)
 {
     // purposely left empty
 }
```

Simple InventoryItem

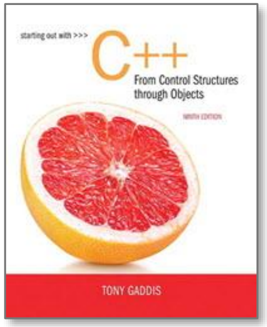
- A simple object to describe an InventoryItem would include:
  - A text description of the item.
  - A cost of the item.
    - A price might also be included (not in book).
  - The number of units.
    - A SKU (stock keeping unit) for the item.



```cpp
// This class has overloaded constructors.
#ifndef INVENTORYITEM_H
#define INVENTORYITEM_H
#include <string>
using namespace std;

class InventoryItem
{
private:
   string description; // The item description
   double cost;        // The item cost
   int units;          // Number of units on hand
public:
   // Constructor #1
   InventoryItem()
      { // Initialize description, cost, and units.
        description = "";
        cost = 0.0;
        units = 0; }

   // Constructor #2
   InventoryItem(string desc)
      { // Assign the value to description.
        description = desc;

        // Initialize cost and units.
        cost = 0.0;
        units = 0; }
```

Continues...

```

   // Constructor #3
   InventoryItem(string desc, double c, int u)
     { // Assign values to description, cost, and units.
       description = desc;
       cost = c;
       units = u; }

   // Mutator functions
   void setDescription(string d)
      { description = d; }

   void setCost(double c)
      { cost = c; }

   void setUnits(int u)
      { units = u; }

   // Accessor functions
   string getDescription() const
      { return description; }

   double getCost() const
      { return cost; }

   int getUnits() const
      { return units; }
};
#endif
```



3.11

Using Private Member Functions

Section 10.2 Exercises

- Can you answer the following:
  - Describe the difference between a class and a structure?
  - Explain why member variables are usually private?
  - Describe the purpose and usage of a constructor?
  - Use an initialization section in a constructor?
  - What is the purpose of a default constructor and when is one supplied automatically in a class.

Answers to Exercises

- Describe the difference between a class and a structure?
  - structs are public by default; classes are private by default.
- Explain why member variables are usually private?
  - Protect programs from class changes and objects from internal corruption.
- Describe purpose and usage of a constructor?
  - Constructors are invoked when an object is created.
  - Used to declare variables.

Answers to Exercisess

- Use initialization section in a constructor?
  - (See slide 98, see slide 111)

```
DayOfYear::DayOfYear(int m, int d) : month(m),
day(d) { }
DayOfYear::DayOfYear() : month(1), day(1) { }
```

- What is the purpose and when is a default constructor provided automatically?
  - If a class does not provide a constructor, a default constructor will automatically be provided that does nothing.
  - A default constructor is needed to declare objects.

As practice consider the following

Take our DayOfYear class and add two constructors: one that takes the month and day to initialize both data members and a default constructor.

Even though, we already added a member function called isEquals, try it on your own. Remember it takes a DayOfYear object as parameter.

Add a member function called isAfter that compares the invoking object to an object passed in as a parameter. if the calling object is later in the year, then return true. Otherwise false.

```
class DayOfYear {
   public:
    int getMonth();
    int getDay();
    void output(ostream& outs);
    void input(istream& ins);
   private:
    void check_date();
    int month;
    int day;
};
```

Start with the class DayOfYear

- On the previous slide is the class DayOfYear before you make your changes.
- The slides that follow show changes
  - Add the two constructors OR
  - Add one constructor with default values.
    - But this is equivalent to 3 constructors:
    - DayOfYear();   // default month = 1, day=1
    - DayOfYear(int mon); // default day = 1
    - DayOfYear(int mon, int day);

```
class DayOfYear {
   public:
    DayOfYear();
    DayOfYear(int mon, int day);
    int getMonth();
    int getDay();
    void output(ostream& outs);
    void input(istream& ins);
   private:
    void check_date();
    int month,day;
};
```

```
class DayOfYear {
   public:
    DayOfYear(int mon=1, int day=1);
    int getMonth();
    int getDay();
    void output(ostream& outs);
    void input(istream& ins);
   private:
    void check_date();
    int month,day;
};
```

Implementation of Constructor

```
// Set data members in initialization
// section, check_date() exits if not
// valid
DayOfYear::DayOfYear(int m, int d) :
month(m), day(d)
{
    check_date();
}
```

Not necessary to use initialization section!

Overloaded constructors

- Remember Rectangle class with the following constructors:

```
Rectangle();
Rectangle(double side);
Rectangle(double w, double l);
```

- What if we added?

```
Rectangle(double w); // Compiler error
```

Overloaded functions must have different number and/or types of parameters!!!

isAfter() member function

- It's tempting to pass an int month and int day, but let's compare to another DayOfYear!

```
// Add to public part of class
   bool isAfter(DayOfYear d);
// Implementation outside of the class
   bool DayOfYear::isAfter(DayOfYear d)
  {
     return (getMonth() > d.getMonth() ||
        (getMonth() == d.getMonth() &&
         getDay() > d.getDay()));
  }
```

Practice Finding the errors.

```
class Automobile {
  public:
      void setPrice(double price);
      void setProfit(double profit);
      double get_price();
  private:
      double get_profit();
      double price;
      double profit;
}; Automobile jaguar, hyundai;
hyundai.price=4999.99; jaquar.set_price(59,000.00);
double a_price = jaguar.get_price(), a_profit = hyundai.get_profit();
if (hyundai == jaguar)
      cout << “Want to swap cars?” << endl;
hyundai = jaguar;
```

See Errors

- If a data member or member function is private, it cannot be accessed outside of the class.
  - Errors are in red.

Practice Finding the errors.

```
class Automobile {
  public:
      void set_price(double price);
      void set_profit(double profit);
      double get_price();
  private:
      double get_profit();
      double price;
      double profit;
}; Automobile jaguar, hyundai;
hyundai.price=4999.99; jaquar.set_price(59,000.00);
double a_price = jaguar.get_price(), a_profit = hyundai.get_profit();
if (hyundai == jaguar) // No operator == exists for Automobile
      cout << “Want to swap cars?” << endl;
hyundai = jaguar;
```

Consider a BankAccount class

- First ask, what is it?
  - Answering "what it is?" gives us the data
  - Could be several data representations

    E.g. a rational number could be...

    3.1459 or 2/3 but is a fraction.

- Then ask, what does it do?
  - Answering "what it does?" gives us the functions.

BankAccount

- What it is?
  - A balance (but that balance can be expressed as…)
    - double balance; // Fractional cents problem
    - int cents;   // More exact
    - int dollars, int cents; // Multiple components
  - An interest rate
  - An account number
    - unsigned long acctNumber;
    - string acctNumber; // Mix of chars, digits

BankAccount

- When a BankAccount is created, there is an initial deposit.
  - BankAccount(int dollars = 0, int cents = 0);
- Typical banking functions include:
  - makeDeposit(double amount)
  - makeWithdrawl(double amount)
  - makeTransfer(double amnt, BankAccount ba)
  - addAccruedInterest(double rate)
  - issueMonthlyStatement()

BankAccount con't

- In order to issue a monthly statement, we need to keep track of transactions.
- An easy way to do that is to create a Transaction structure that is used by the BankAccount object.
- Private data for BankAccount includes:
  - double balance;
  - double interest_rate;
  - vector <Transaction> transactions;

struct transaction

- Every transaction has a dollar amount.
- Every transaction has a date (and time).
- Every transaction has a type.
  - We'll use a similar construct as wage_status in a previous example.
- There are 5 different transaction types:
  - Deposits
  - Withdrawls
  - Interest payments

  Transfers

  Overdraft

Struct transaction

- Here’s the struct that represents Transaction type in our BankAccount application.
- struct Transaction {

```
   char    type;          // ‘D’, ‘W’, ‘O’, ‘I’, ‘T’
   double amount;
   Date    date;          // Date struct or class
 };
```

- In order to issue a monthly statement, we need to keep a list of Transactions.

struct Transaction

- How do we keep track of transactions for the month?
  - vector<Transaction> transactions;
  - Write a method to shrink_wrap or initialize a transaction to be stored in transactions.
- How do we get a month's worth of transactions?
  - Write a method to start at the end and go backwards until we hit the previous month.

Class BankAccount

- Remember that we want BankAccount to support normal banking functions such as:
  - makeDeposit(double amount)
  - makeWithdrawl(double amount)
  - makeTransfer(double amnt, BankAccount ba)
  - addAccruedInterest(double rate)
  - issueMonthlyStatement(DayOfYear from)
  - getBalance(), getInterestRate(), getAccountNum()
- Each banking function will be represented by a member function.

Class BankAccount and struct Transaction

```
/*
 * BankAccount class
 *    keeps the data along with all of the functions that apply to a BankAccount.
 */
class BankAccount {
public:
    BankAccount(int dollars = 0, int cents = 0); // Includes default constructor
// Constants that apply to the whole class
    static const char DEPOSIT = 'D';
    static const char WITHDRAWL = 'W’;
    static const char INTEREST = 'I';
    static const char OVERDRAFT = ‘O’;
    static const char TRANSFER = ‘T’;
// Accessors/Mutators (setters/getters)
    void set(double balance, double interest_rate);
    double getBalance();
    double getInterestRate();
```

Class BankAccount and struct Transaction

```
    unsigned long getAccountNum();
// Public method interface
    double makeDeposit(double deposit);
    double makeWithdrawl(double withdrawl);
    double makeTransfer(double transfer, BankAccount toAcct);
    double addAccruedInterest();
    void issueMonthlyStatement(DayOfYear from, DayOfYear until);
private:
// Private utility methods
    Transaction shrink_wrap(char type, double amount, time_t date=0);
    void record_transaction(unsigned long account, char type, double amount, Date td);
// Instance variables
    unsigned long account_num;           // Accommodates a long number
    int cents;                           // No danger of fractional cents.
    double interest_rate;
    vector<Transaction> transactions;    // Audit trail of all transactions.
};
```

10.3

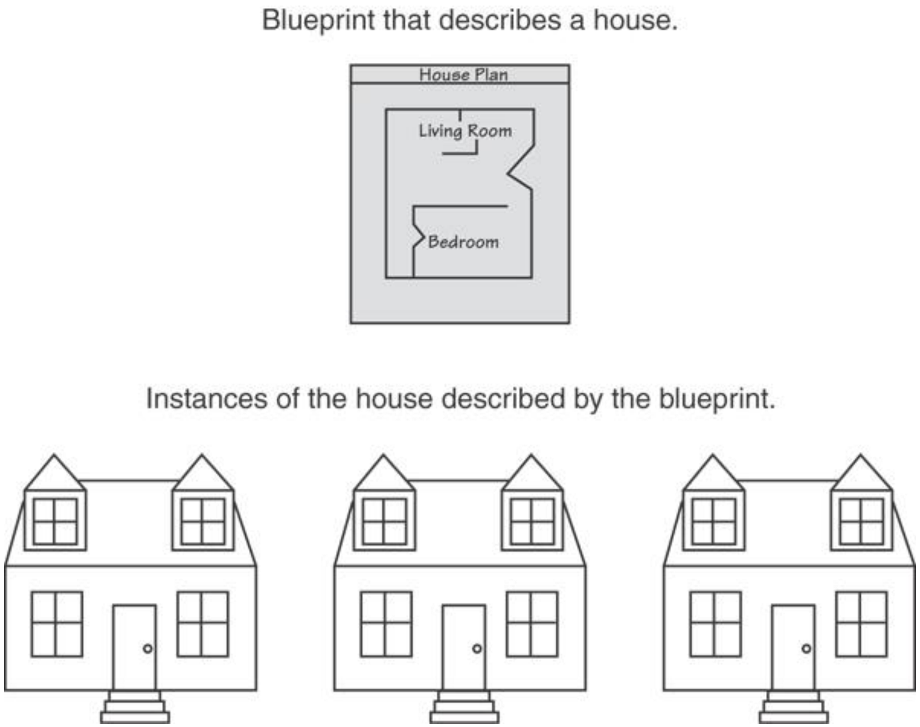
Abstract Data Types

Object-Oriented Programming Terminology

- class: like a `struct` (allows bundling of related variables), but variables and functions in the class can have different properties than in a `struct`
- object: an instance of a `class`, in the same way that a variable can be an instance of a `struct`

Classes and Objects

- A Class is like a blueprint and objects are like houses (instances) built from the blueprint



Blueprint that describes a house.

Instances of the house described by the blueprint.

Object-Oriented Programming Terminology

- properties: data members of a class aka instance variables.
- methods or behaviors: member functions of a class are known as methods.

- invoking object is the calling object.
  - When a method is invoked there is always an invoking object that is copied into a special pointer known as the **this** pointer

Abstract Data Types

- A data type consists of a collection of values together with a set of basic operations defined on the values
  - example: int type and its associated valid operations
- A data type is an **Abstract Data Type (ADT)** if programmers using the type do not have access to the details of how the values and operations are implemented
  - example: int, double

Procedural and Object-Oriented Programming

- Procedural programming focuses on the process/actions that occur in a program

- Object-Oriented programming is based on the data and the functions that operate on it. Objects are instances of ADTs that represent the data and its functions

Classes To Produce ADTs

- To define a class so it is an ADT
  - Separate the specification of how the type is used by a programmer from the details of how the type is implemented
  - Make all member variables **private** members
  - Helper functions should be private members
  - Basic operations a programmer needs should be **public** member functions
  - Fully specify how to use each public function

More on Objects

- data hiding: restricting access to certain members of an object
  - Protects programs from changes in classes.
  - Protects objects from data corruption.

- public interface: members of an object that are available outside of the object. This allows the object to provide access to some data and functions without sharing its internal details and design, and provides some protection from data corruption

ADT Interface

- The ADT interface tells how to use the ADT in a program
  - The interface consists of
    - The public member functions' declarations or prototypes
    - The comments that explain how to use those functions
  - The interface should be all that is needed to know how to use the ADT in a program

ADT Implementation

- The ADT implementation tells how the interface is **realized** in C++
  - The **implementation** consists of
    - The private members of the class
    - The definitions of public and private member functions
  - The implementation of a class's interface is needed to run a program that uses the class.
  - The implementation is not needed to write the main part of a program or any non-member functions

ADT Benefits

- Changing an ADT implementation does not require changing a program that uses the ADT
- ADT's make it easier to divide work among different programmers
  - One or more can write the ADT
  - One or more can write code that uses the ADT
- Writing and using ADTs breaks the larger programming task into smaller tasks

Program Example
The BankAccount ADT

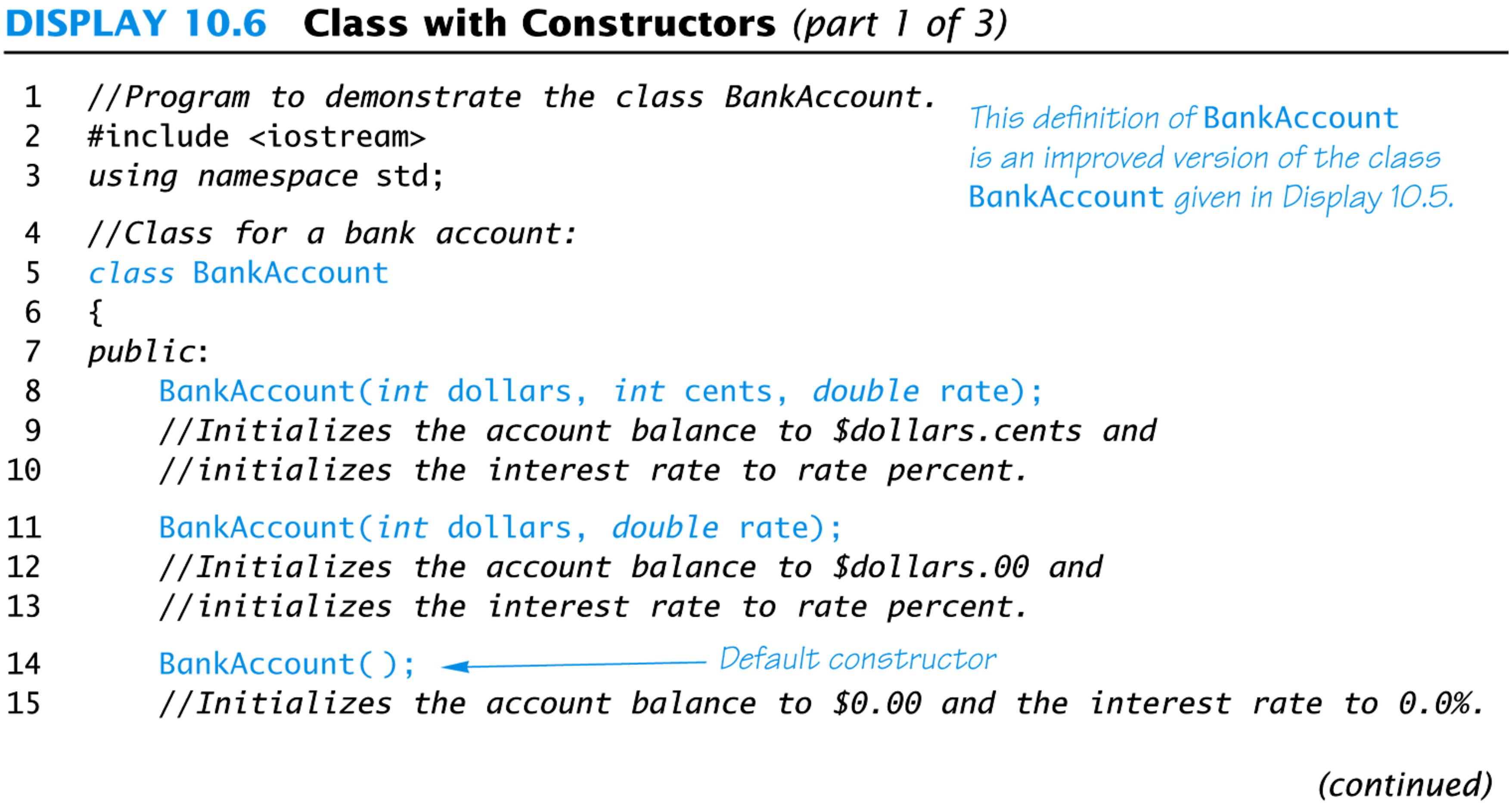
- In the version of the BankAccount ADT shown in Display 10.7.
  - Data is stored as three member variables
    - The dollars part of the account balance
    - The cents part of the account balance
    - The interest rate
  - This version stores the interest rate as a fraction
  - The public portion of the class definition remains unchanged from the version of Display 10.6

Same interface, different implementation

Display 10.6

Display 10.7

Display 10.6 (1/3)



DISPLAY 10.6 Class with Constructors *(part 1 of 3)*

```
//Program to demonstrate the class BankAccount.
#include <iostream>
using namespace std;

//Class for a bank account:
class BankAccount
{
public:
    BankAccount(int dollars, int cents, double rate);
    //Initializes the account balance to $dollars.cents and
    //initializes the interest rate to rate percent.

    BankAccount(int dollars, double rate);
    //Initializes the account balance to $dollars.00 and
    //initializes the interest rate to rate percent.

    BankAccount( );
    //Initializes the account balance to $0.00 and the interest rate to 0.0%.
```

This definition of BankAccount *is an improved version of the class* BankAccount *given in Display 10.5.*

Default constructor

(continued)

DISPLAY 10.6 Class with Constructors *(part 2 of 3)*

```
     void update( );
    //Postcondition: One year of simple interest has been added to the account
    //balance.

    double get_balance( );
    //Returns the current account balance.

    double get_rate( );
    //Returns the current account interest rate as a percentage.

    void output(ostream& outs);
    //Precondition: If outs is a file output stream, then
    //outs has already been connected to a file.
    //Postcondition: Account balance and interest rate have been written to the
    //stream outs.
private:
    double balance;
    double interest_rate;

    double fraction(double percent);
    //Converts a percentage to a fraction. For example, fraction(50.3)
    //returns 0.503.
};

int main( )
{
    BankAccount account1(100, 2.3), account2;

    cout << "account1 initialized as follows:\n";
    account1.output(cout);
    cout << "account2 initialized as follows:\n";
    account2.output(cout);

    account1 = BankAccount(999, 99, 5.5);
    cout << "account1 reset to the following:\n";
    account1.output(cout);
    return 0;
}

BankAccount::BankAccount(int dollars, int cents, double rate)
{
    if ((dollars < 0) || (cents < 0) || (rate < 0))
    {
        cout << "Illegal values for money or interest rate.\n";
        exit(1);
    }
```

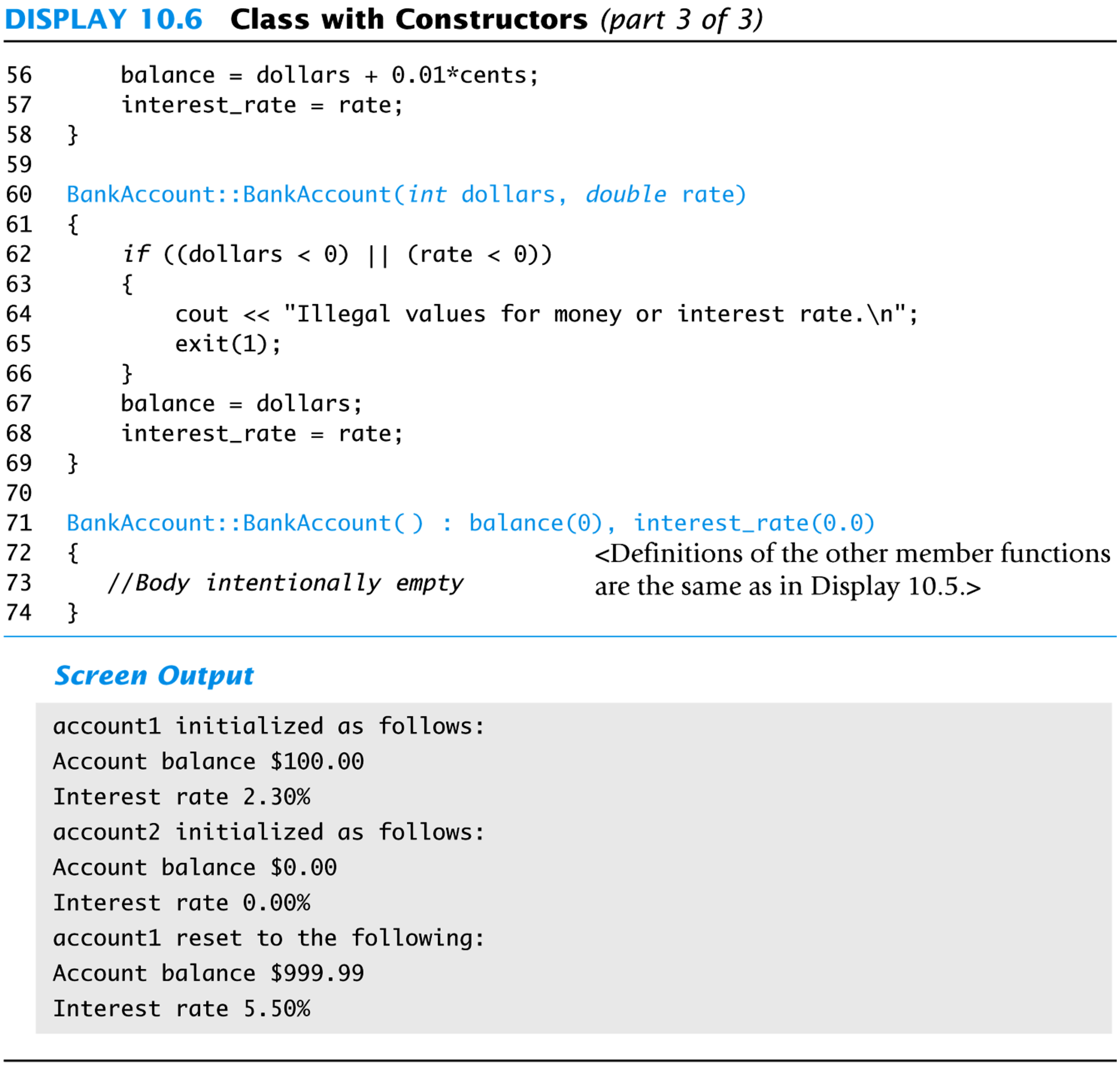
This declaration causes a call to the default constructor. Notice that there are no parentheses.

An explicit call to the constructor `BankAccount::BankAccount`

(continued)

Display 10.6 (2/3)

Display 10.6 (3/3)



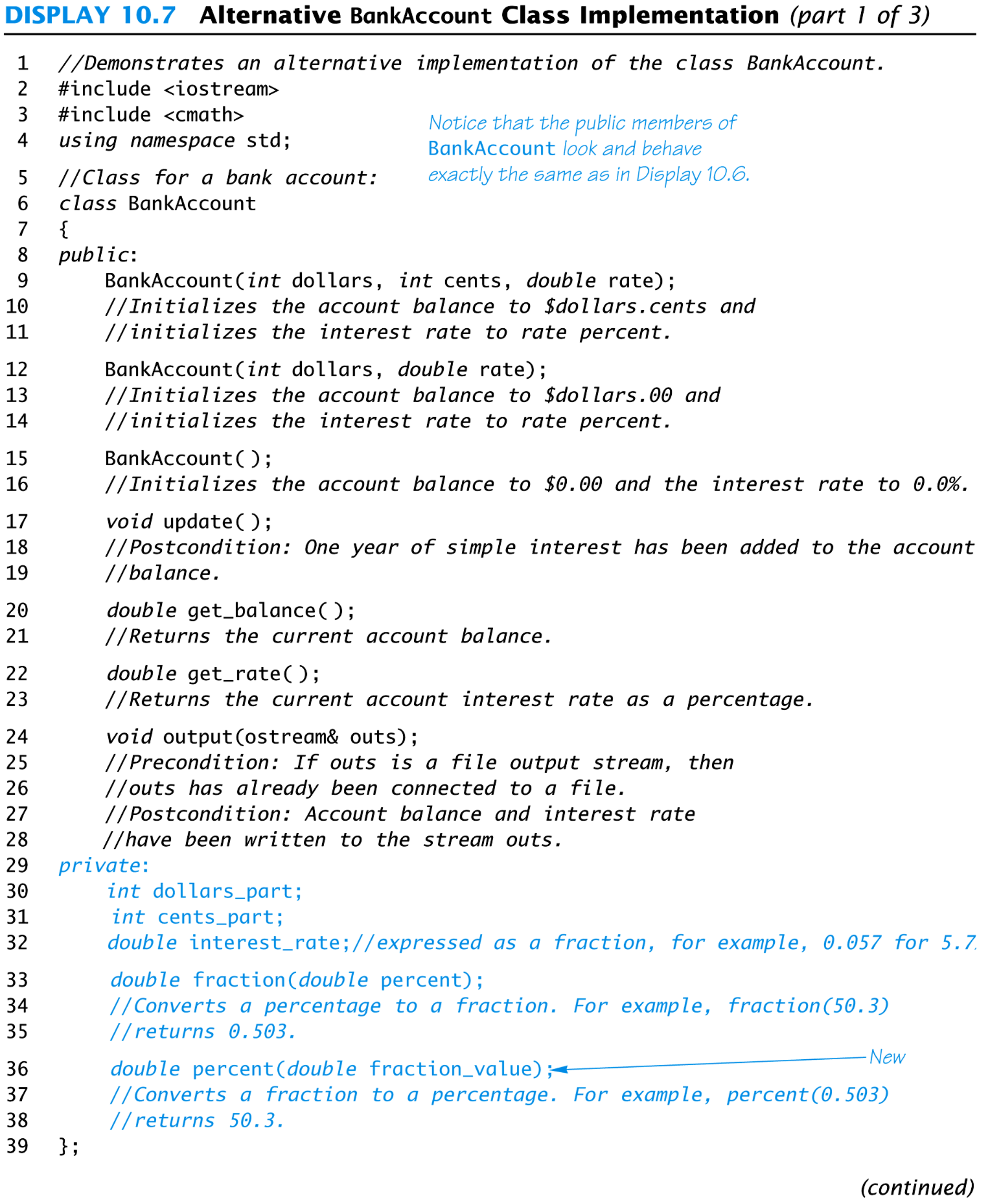
DISPLAY 10.6 Class with Constructors *(part 3 of 3)*

```
    balance = dollars + 0.01*cents;
    interest_rate = rate;
}

BankAccount::BankAccount(int dollars, double rate)
{
    if ((dollars < 0) || (rate < 0))
    {
        cout << "Illegal values for money or interest rate.\n";
        exit(1);
    }
    balance = dollars;
    interest_rate = rate;
}

BankAccount::BankAccount( ) : balance(0), interest_rate(0.0)
{
   //Body intentionally empty
}
```

<Definitions of the other member functions are the same as in Display 10.5.>

Screen Output

```
account1 initialized as follows:
Account balance $100.00
Interest rate 2.30%
account2 initialized as follows:
Account balance $0.00
Interest rate 0.00%
account1 reset to the following:
Account balance $999.99
Interest rate 5.50%
```



DISPLAY 10.7 Alternative BankAccount Class Implementation *(part 1 of 3)*

```
//Demonstrates an alternative implementation of the class BankAccount.
#include <iostream>
#include <cmath>
using namespace std;
//Class for a bank account:
class BankAccount
{
public:
    BankAccount(int dollars, int cents, double rate);
    //Initializes the account balance to $dollars.cents and
    //initializes the interest rate to rate percent.
    BankAccount(int dollars, double rate);
    //Initializes the account balance to $dollars.00 and
    //initializes the interest rate to rate percent.
    BankAccount( );
    //Initializes the account balance to $0.00 and the interest rate to 0.0%.
    void update( );
    //Postcondition: One year of simple interest has been added to the account
    //balance.
    double get_balance( );
    //Returns the current account balance.
    double get_rate( );
    //Returns the current account interest rate as a percentage.
    void output(ostream& outs);
    //Precondition: If outs is a file output stream, then
    //outs has already been connected to a file.
    //Postcondition: Account balance and interest rate
    //have been written to the stream outs.
private:
    int dollars_part;
    int cents_part;
    double interest_rate;//expressed as a fraction, for example, 0.057 for 5.7
    double fraction(double percent);
    //Converts a percentage to a fraction. For example, fraction(50.3)
    //returns 0.503.
    double percent(double fraction_value);
    //Converts a fraction to a percentage. For example, percent(0.503)
    //returns 50.3.
};
```

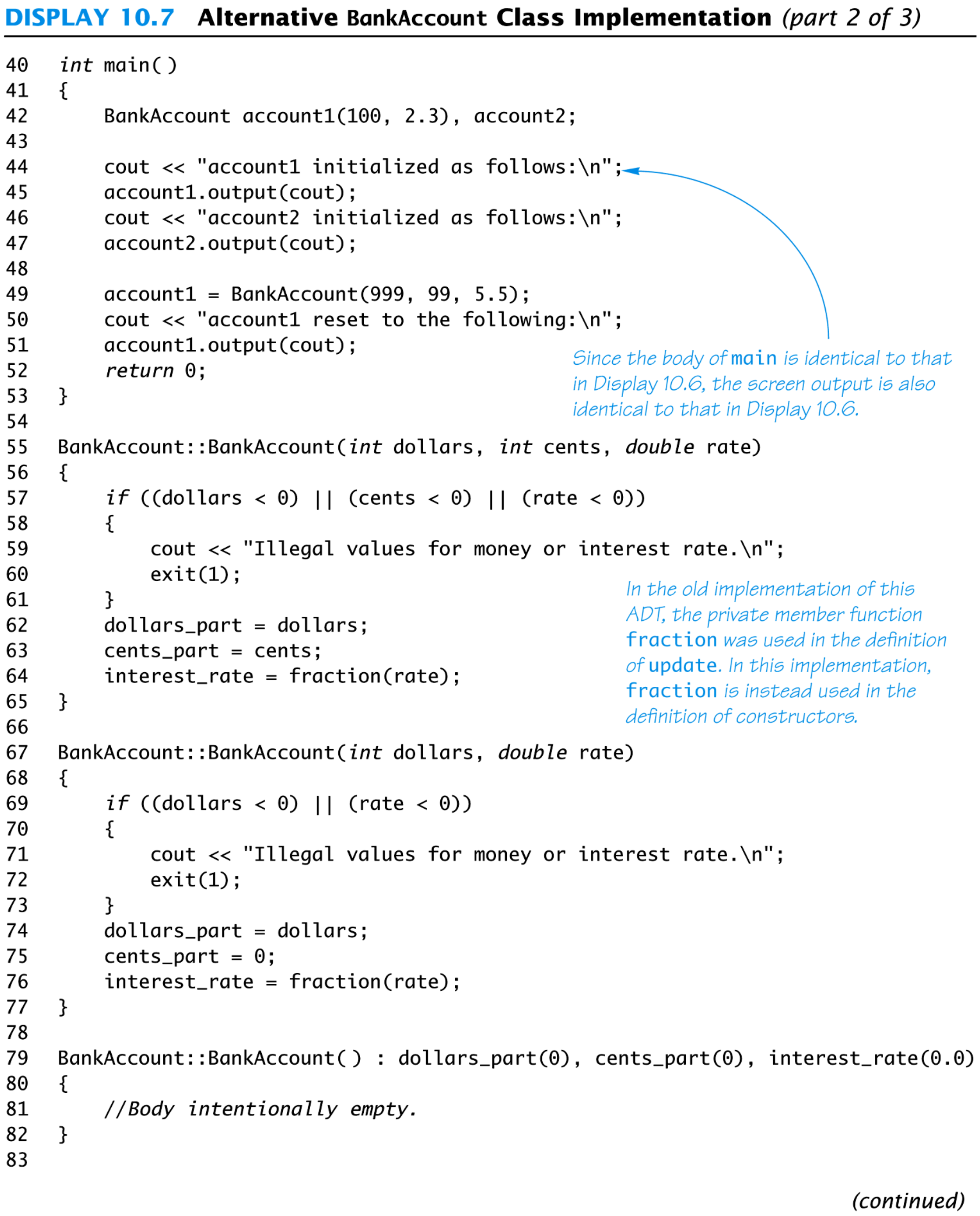
Notice that the public members of BankAccount look and behave exactly the same as in Display 10.6.

New

(continued)

Display 10.7 (1/3)

Display 10.7 (2/3)



DISPLAY 10.7 Alternative BankAccount Class Implementation *(part 2 of 3)*

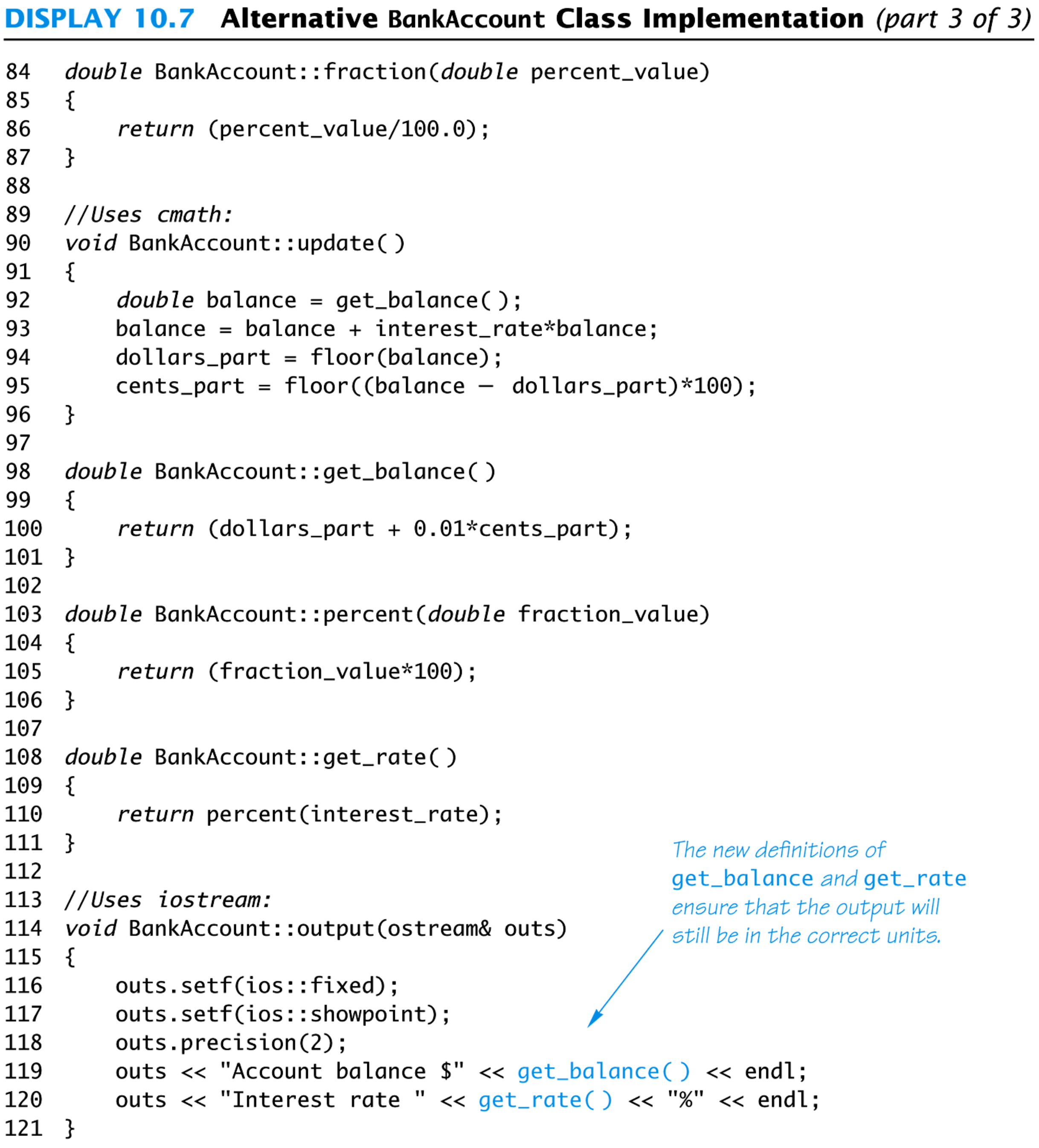
```
int main( )
{
    BankAccount account1(100, 2.3), account2;

    cout << "account1 initialized as follows:\n";
    account1.output(cout);
    cout << "account2 initialized as follows:\n";
    account2.output(cout);

    account1 = BankAccount(999, 99, 5.5);
    cout << "account1 reset to the following:\n";
    account1.output(cout);
    return 0;
}

BankAccount::BankAccount(int dollars, int cents, double rate)
{
    if ((dollars < 0) || (cents < 0) || (rate < 0))
    {
        cout << "Illegal values for money or interest rate.\n";
        exit(1);
    }
    dollars_part = dollars;
    cents_part = cents;
    interest_rate = fraction(rate);
}

BankAccount::BankAccount(int dollars, double rate)
{
    if ((dollars < 0) || (rate < 0))
    {
        cout << "Illegal values for money or interest rate.\n";
        exit(1);
    }
    dollars_part = dollars;
    cents_part = 0;
    interest_rate = fraction(rate);
}

BankAccount::BankAccount( ) : dollars_part(0), cents_part(0), interest_rate(0.0)
{
    //Body intentionally empty.
}

```

Since the body of **main** *is identical to that in Display 10.6, the screen output is also identical to that in Display 10.6.*

In the old implementation of this ADT, the private member function **fraction** *was used in the definition of* **update**. *In this implementation,* **fraction** *is instead used in the definition of constructors.*

(continued)

Display 10.7 (3/3)



DISPLAY 10.7 Alternative BankAccount Class Implementation *(part 3 of 3)*

```
double BankAccount::fraction(double percent_value)
{
    return (percent_value/100.0);
}

//Uses cmath:
void BankAccount::update( )
{
    double balance = get_balance( );
    balance = balance + interest_rate*balance;
    dollars_part = floor(balance);
    cents_part = floor((balance -  dollars_part)*100);
}

double BankAccount::get_balance( )
{
    return (dollars_part + 0.01*cents_part);
}

double BankAccount::percent(double fraction_value)
{
    return (fraction_value*100);
}

double BankAccount::get_rate( )
{
    return percent(interest_rate);
}

//Uses iostream:
void BankAccount::output(ostream& outs)
{
    outs.setf(ios::fixed);
    outs.setf(ios::showpoint);
    outs.precision(2);
    outs << "Account balance $" << get_balance( ) << endl;
    outs << "Interest rate " << get_rate( ) << "%" << endl;
}
```

The new definitions of **get_balance** *and* **get_rate** *ensure that the output will still be in the correct units.*

Interface Preservation

- To preserve the interface of an ADT so that programs using it do not need to be changed
  - Public member declarations cannot be changed
  - Public member **definitions** (i.e., **implementation** or realization) can be changed
  - Private member functions can be added, deleted, or changed

Information Hiding

- Information hiding was referred to earlier as writing functions so they can be used like black boxes
- ADT's does information hiding because
  - The interface is all that is needed to use the ADT
  - Implementation details of the ADT are not needed to know how to use the ADT
  - Implementation details of the data values are not needed to know how to use the ADT

Section 10.3 Exercises

- Can you
  - Describe an ADT?
  - Describe how to implement an ADT in C++?
  - Define the interface of an ADT?
  - Define the implementation of an ADT?

Another example

- Let's create an ADT called Rectangle.

- When we define an ADT we have to think about what it needs to do and what it needs to be.

- What an ADT needs to be is defined by the member variables.

- What an ADT needs to do is defined by the member functions.

A Rectangle needs to be…

- A shape with a positive, non-zero length and a height.

- A shape that can have equal length and height which makes it the special case of a Square.

- A shape that can be drawn needs a position and orientation (angle).

Rectangle

- What kinds of things do you do with a Rectangle?
  - Get length and height
  - Calculate Perimeter
  - Calculate Area
  - isSquare
  - hasSamePerimeter
  - hasSameArea
  - Draw, Move, Rotate… // drawing program.

Rectangle constructors

- Define a default constructor

- Define a constructor for the special case of square.

- Define a constructor for initializing both dimensions at creation.

Class Rectangle – public part

```
class Rectangle {
  public:
        Rectangle();                                     //default constructor
        Rectangle(int side);                             // Square
        Rectangle(int length, int height);
        int get_length();                                // length accessor
        int get_height();                                // height accessor
        int calculatePerimeter();
        int calculateArea();
        bool hasSameArea(const Rectangle& r);
        bool hasSamePerimeter(const Rectangle& r);
        bool isSquare();
```

class Rectangle - private

```
class Rectangle {
  public:
        // see previous slide
        void input(const istream& in);
        void output(const ostream& out);
  private:
        int length;                    // member variables
        int height;
        bool check_dimensions();
};
```

10.4

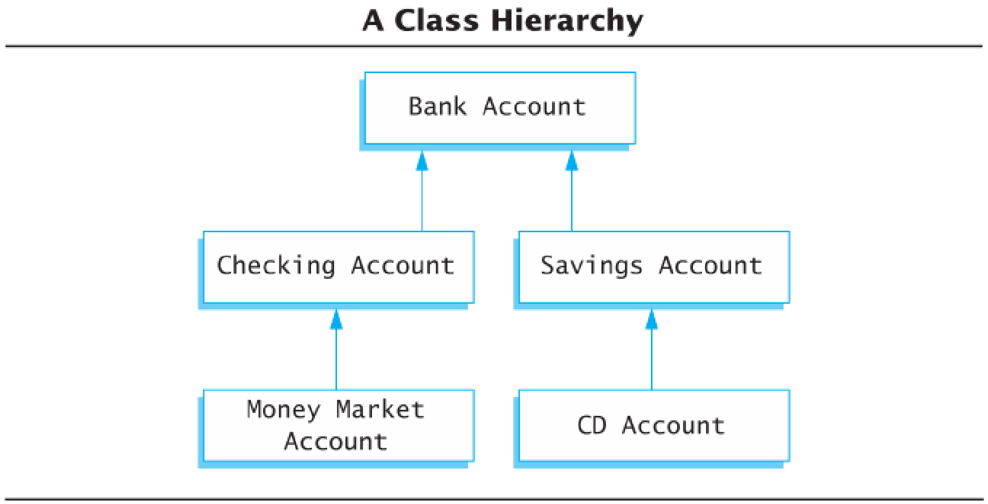
Introduction to Inheritance

Inheritance

- Inheritance refers to derived classes
  - Derived classes are obtained from another class by adding features
  - A derived class inherits the member functions and variables from its parent class without having to re-write them

Inheritance Example

- Natural hierarchy of bank accounts
- Most general: A Bank Account stores a balance
- A Checking Account "IS A" Bank Account that allows customers to write checks
- A Savings Account "IS A" Bank Account without checks but higher interest

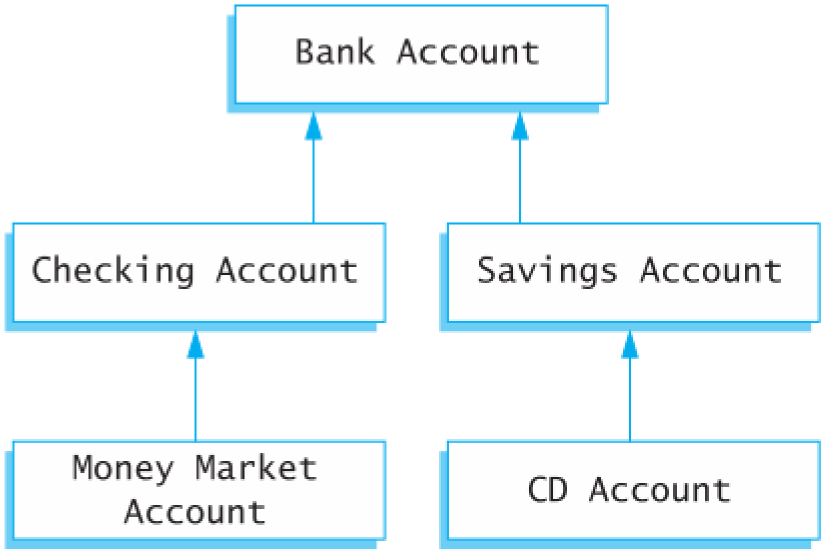


Accounts are more specific as we go down the hierarchy

Each box can be a class

Display 10.8

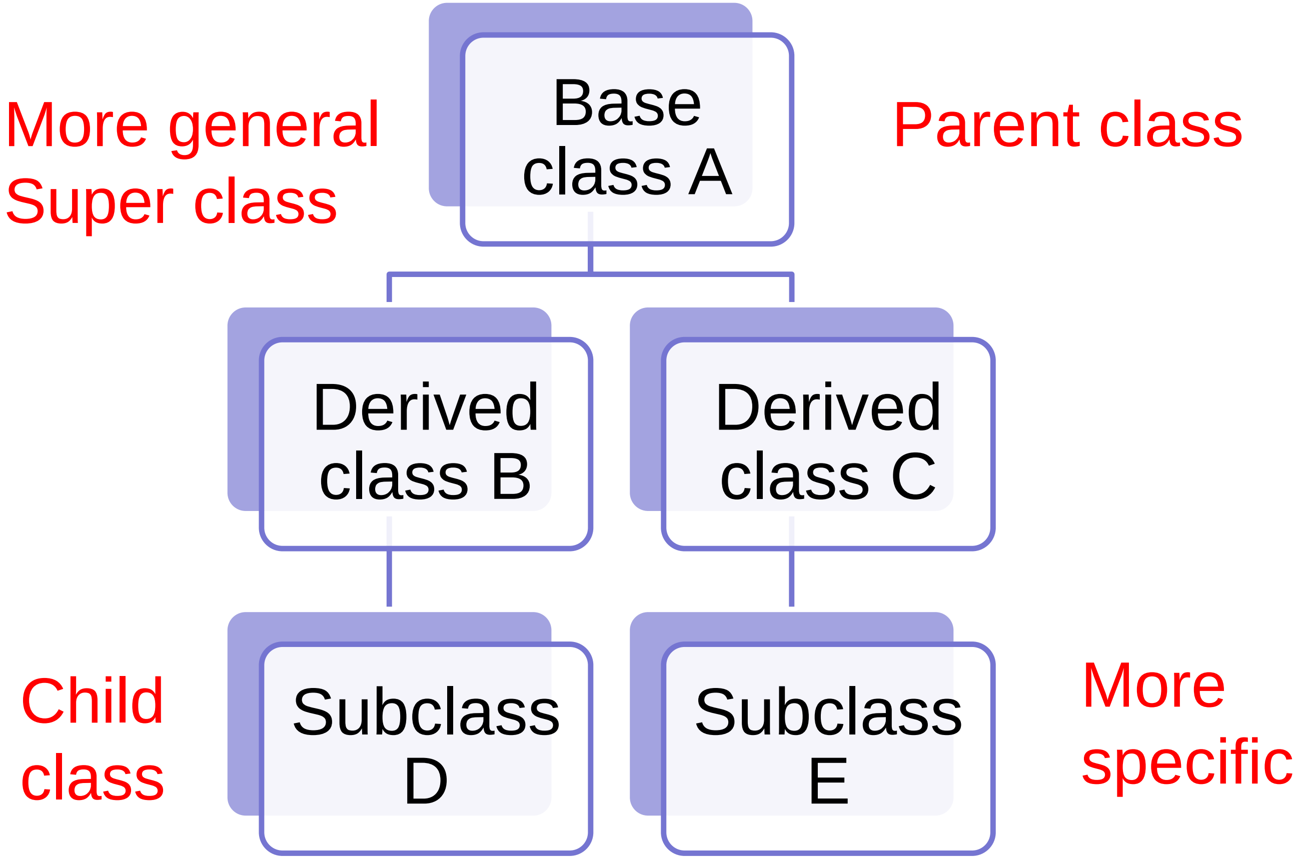
A Class Hierarchy



Inheritance Relationships

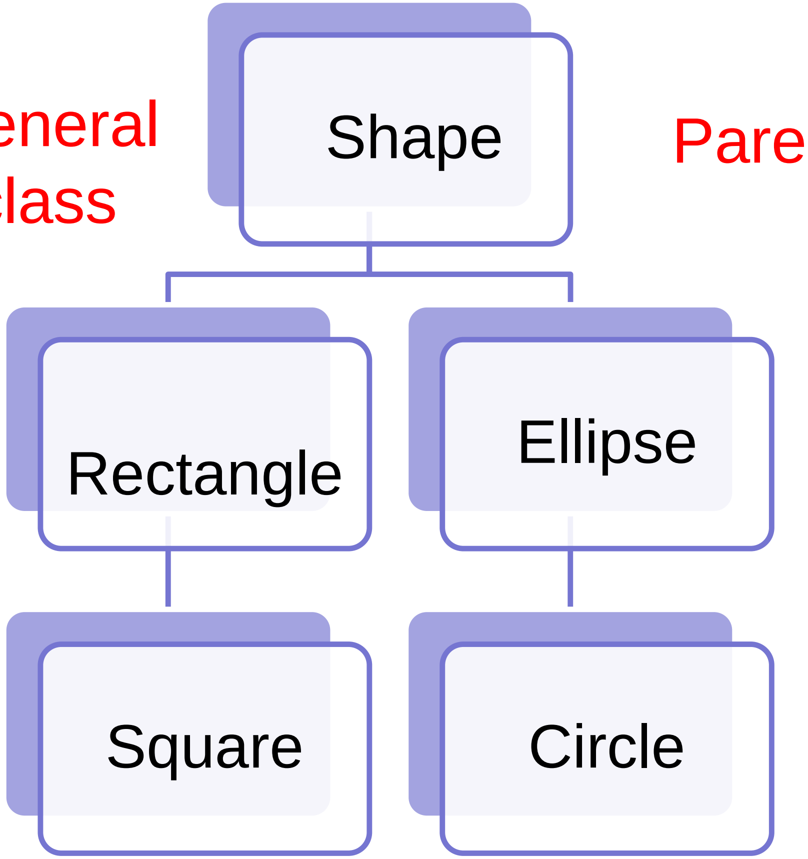
- The more specific class is a **derived** or **child** class
- The more general class is the **base**, **super**, or **parent** class
- If class B is derived from class A
  - Class B is a derived class of class A
  - Class B is a child of class A
  - Class A is the parent of class B
  - Class B inherits the member functions and variables of class A

Inheritance hierarchy



Inheritance hierarchy - general to specific

More general
Super class



Shape

Rectangle

Ellipse

Square

Circle

Parent class

Child
class

More
specific

Define Derived Classes

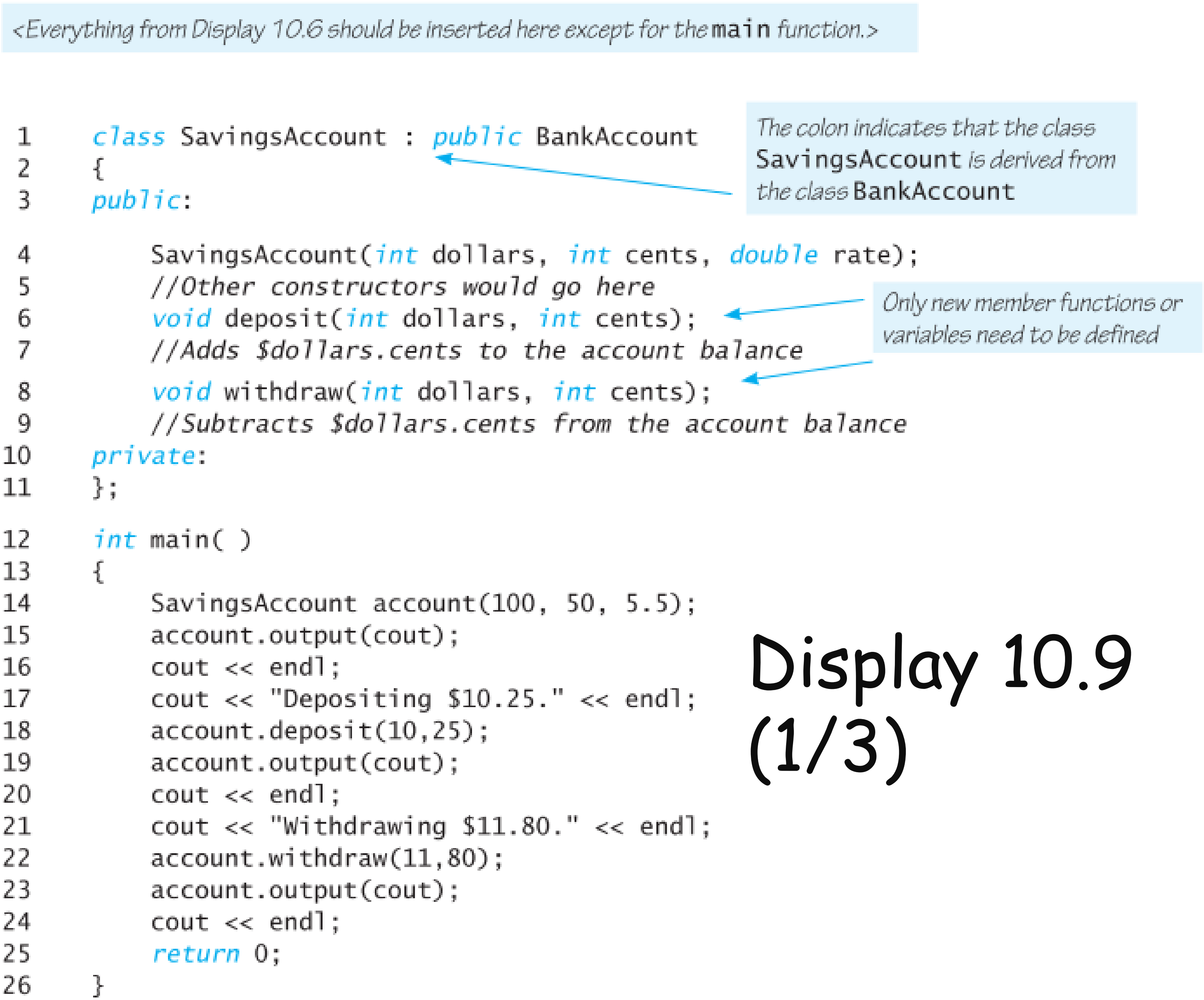
- Give the class name as normal, but add a colon and then the name of the base class

```
class SavingsAccount : public BankAccount
{
   …
}
```

- Objects of type SavingsAccount can access member functions defined in SavingsAccount or BankAccount

Display 10.9 (1-3)

Display 10.9 (1/3)



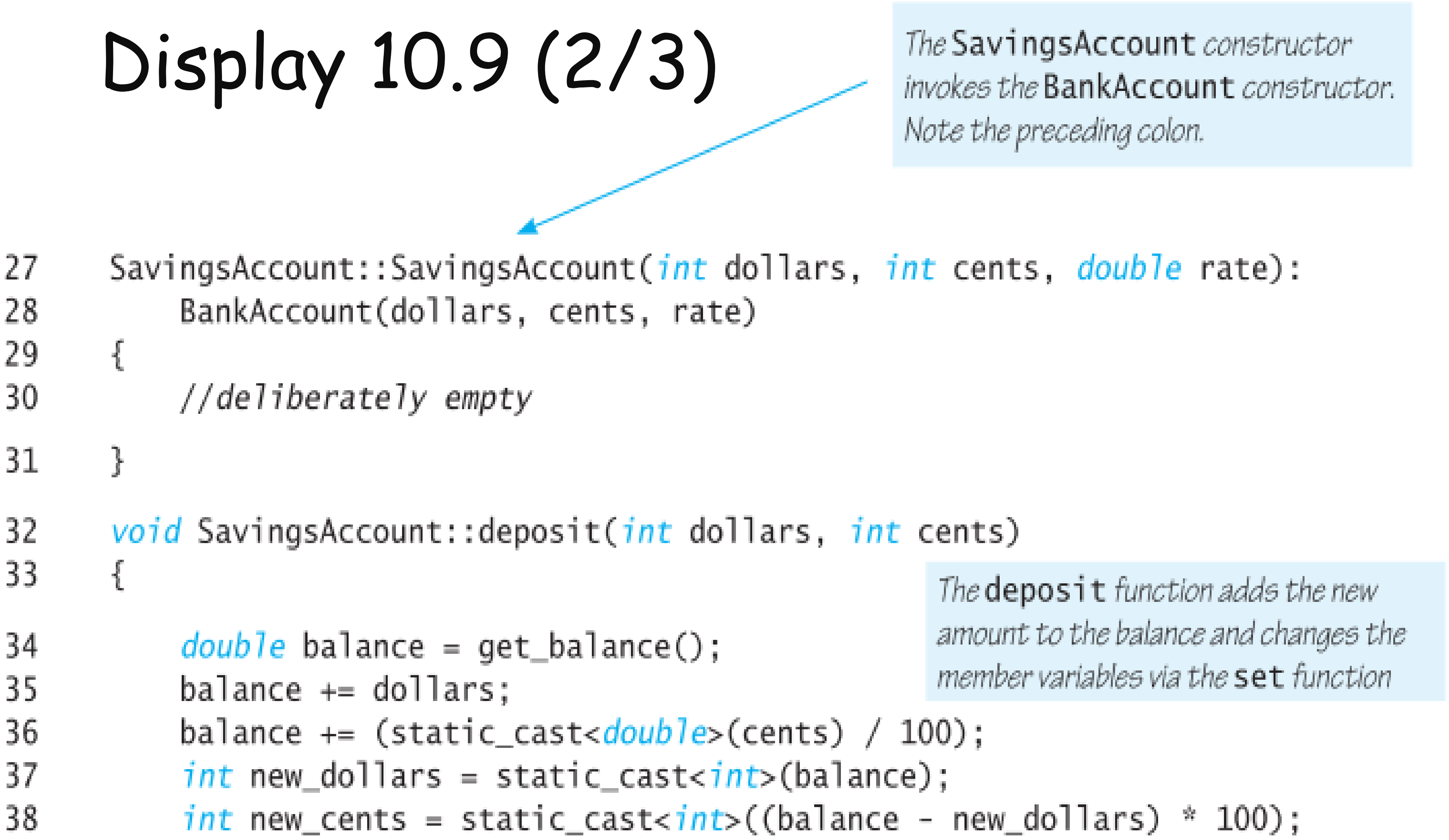
<Everything from Display 10.6 should be inserted here except for the main function.>

```
class SavingsAccount : public BankAccount
{
public:
    SavingsAccount(int dollars, int cents, double rate);
    //Other constructors would go here
    void deposit(int dollars, int cents);
    //Adds $dollars.cents to the account balance

    void withdraw(int dollars, int cents);
    //Subtracts $dollars.cents from the account balance
private:
};

int main( )
{
    SavingsAccount account(100, 50, 5.5);
    account.output(cout);
    cout << endl;
    cout << "Depositing $10.25." << endl;
    account.deposit(10,25);
    account.output(cout);
    cout << endl;
    cout << "Withdrawing $11.80." << endl;
    account.withdraw(11,80);
    account.output(cout);
    cout << endl;
    return 0;
}
```

The colon indicates that the class SavingsAccount is derived from the class BankAccount

Only new member functions or variables need to be defined

Display 10.9 (2/3)

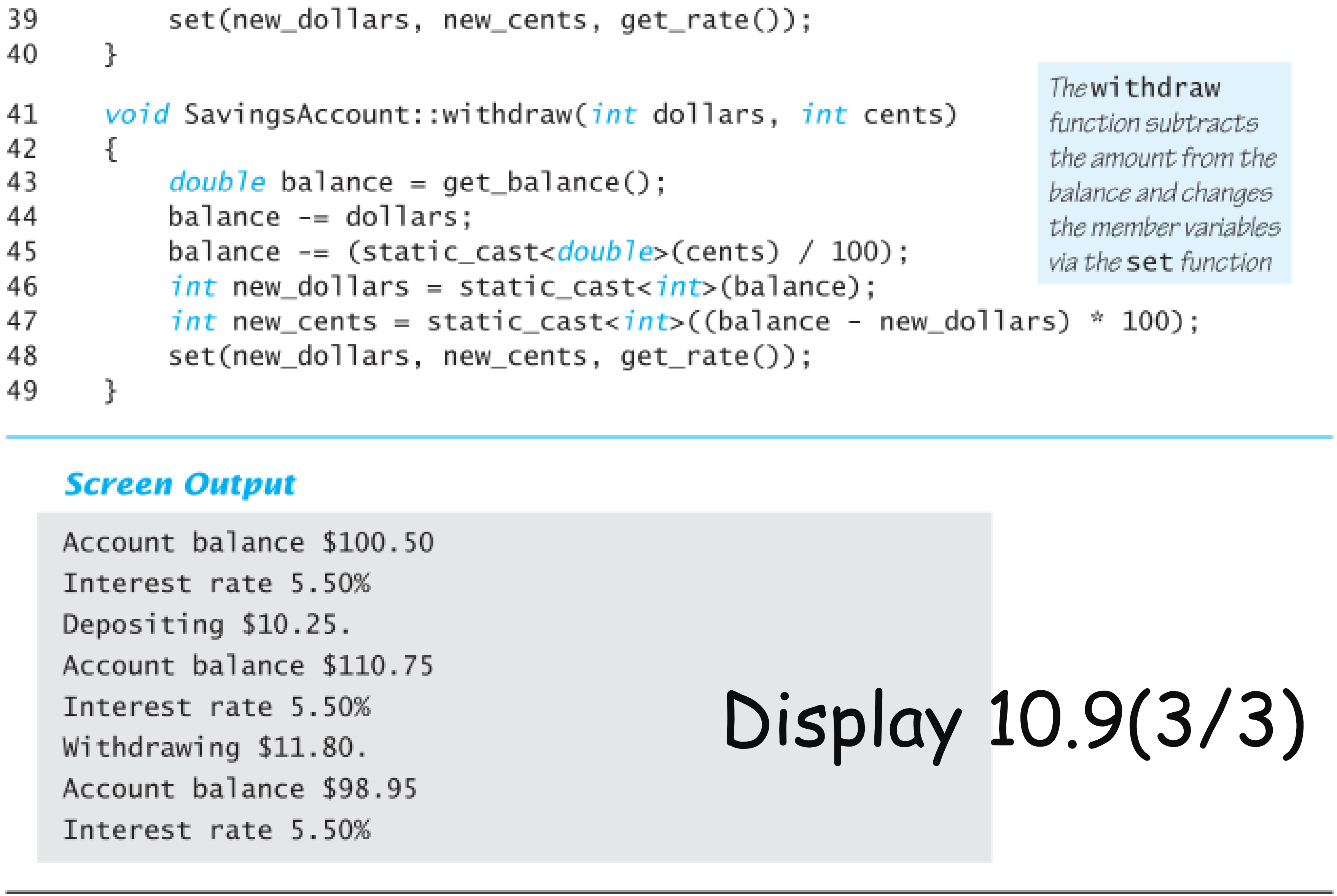


The SavingsAccount constructor invokes the BankAccount constructor. Note the preceding colon.

```
SavingsAccount::SavingsAccount(int dollars, int cents, double rate):
    BankAccount(dollars, cents, rate)
{
    //deliberately empty

}

void SavingsAccount::deposit(int dollars, int cents)
{

    double balance = get_balance();
    balance += dollars;
    balance += (static_cast<double>(cents) / 100);
    int new_dollars = static_cast<int>(balance);
    int new_cents = static_cast<int>((balance - new_dollars) * 100);
```

The deposit function adds the new amount to the balance and changes the member variables via the set function

For more information on type casting, http://www.cplusplus.com/doc/tutorial/typecasting/



```
        set(new_dollars, new_cents, get_rate());
    }

    void SavingsAccount::withdraw(int dollars, int cents)
    {
        double balance = get_balance();
        balance -= dollars;
        balance -= (static_cast<double>(cents) / 100);
        int new_dollars = static_cast<int>(balance);
        int new_cents = static_cast<int>((balance - new_dollars) * 100);
        set(new_dollars, new_cents, get_rate());
    }
```

The `withdraw` *function subtracts the amount from the balance and changes the member variables via the* `set` *function*

Screen Output

```
Account balance $100.50
Interest rate 5.50%
Depositing $10.25.
Account balance $110.75
Interest rate 5.50%
Withdrawing $11.80.
Account balance $98.95
Interest rate 5.50%
```

Display 10.9(3/3)

Section 10.4 Exercises

- Can you
  - Define object?
  - Define class?
  - Describe the relationship between parent and child classes?
  - Describe the benefit of inheritance?