Algorithms and Data Structures in C++

Dr. Robert Amelard

ramelard@uwaterloo.ca

[with material from “Absolute C++”, by Savitch and Mock, Published by Addison Wesley, Copyright Addison Wesley]

Objectives

Core Content:

- **Introduction to Algorithms and Data Structures**
- **Defining C++ Classes**
- **Public and Private Class Members**
- **Accessor and Mutator Functions**
- **Class Constructors and Destructors**

Additional Information:

- Copy Constructors
- Using Inline Functions and Static Member Data
- Operator Overloading
- Declaring const Functions
- Declaring friend Functions

Why Study Algorithms and Data Structures?

- **Why study algorithms and data structures?**
  - One of the most important topics in computing
  - Critical in modern software development
  - Forms the backbone of the world’s most complex software systems that control:
    - Electronic devices, such as smartphones, game consoles, computers, etc
    - Power plants
    - Health monitoring systems
    - Robots, including manufacturing, navigation, etc

- **Example:**
  - Navigation robots and NASA Mars Exploration Rover (MER)

What is an Algorithm? /1

- **Algorithm**
  - A finite sequence of unambiguous instructions performed to achieve a goal or compute a desired result

- **Algorithmics**
  - The study of algorithms

- **Each algorithm is not a solution, but instead a precisely defined procedure for deriving solutions**

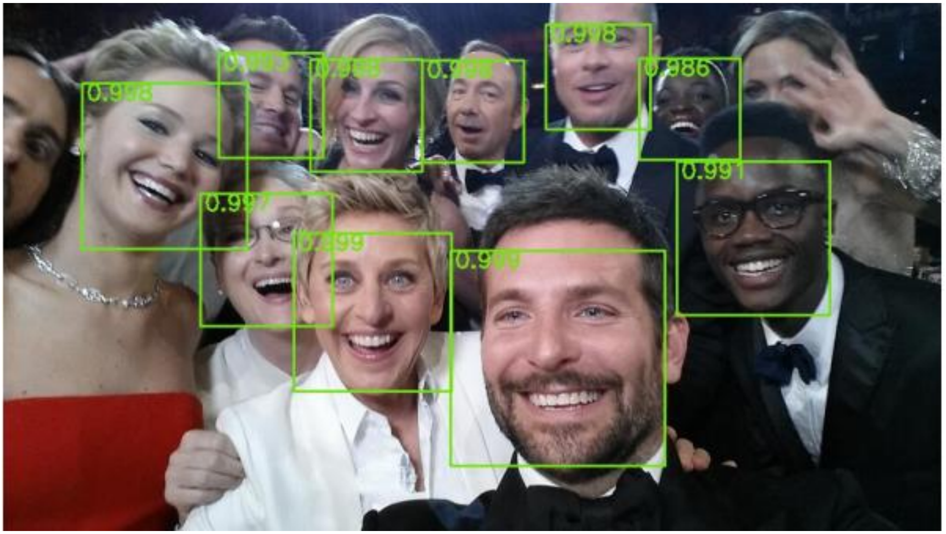
- **Process**
  - A sequence of operations performed to achieve a goal
  - Processes do not have to terminate (e.g., living and breathing)

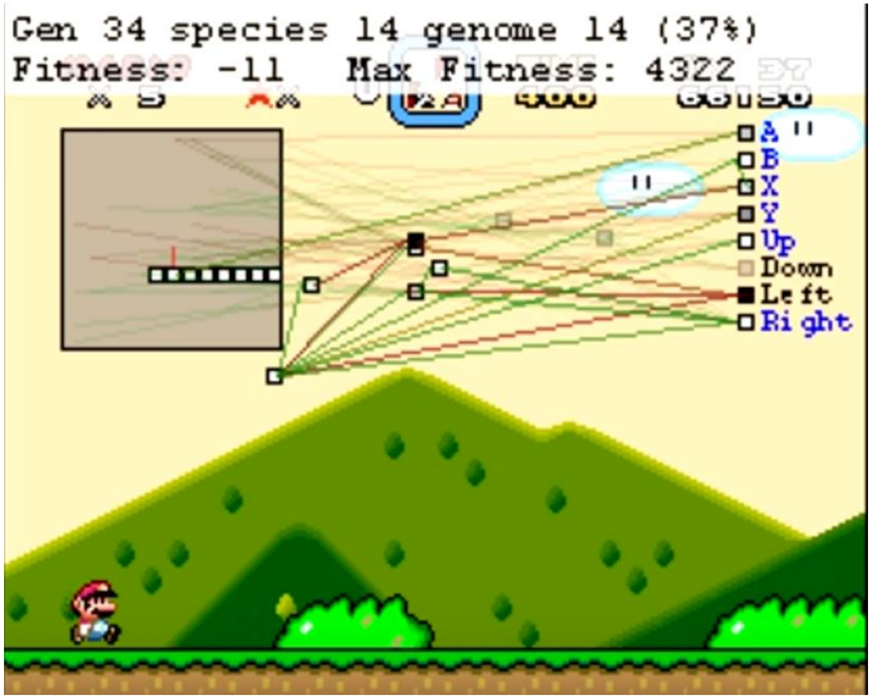
What is an Algorithm? /2

- **Algorithm vs. Process**
  - Algorithm has each step unambiguously specified
  - Process represents higher complexity of work
    - May contain multiple algorithms as steps
  - Process specification may contain ambiguity
    - E.g., Increase customer awareness
  - Algorithm has clearly defined termination
  - Process may be a continually ongoing activity
    - E.g., Lifelong learning

What is an Algorithm? /3

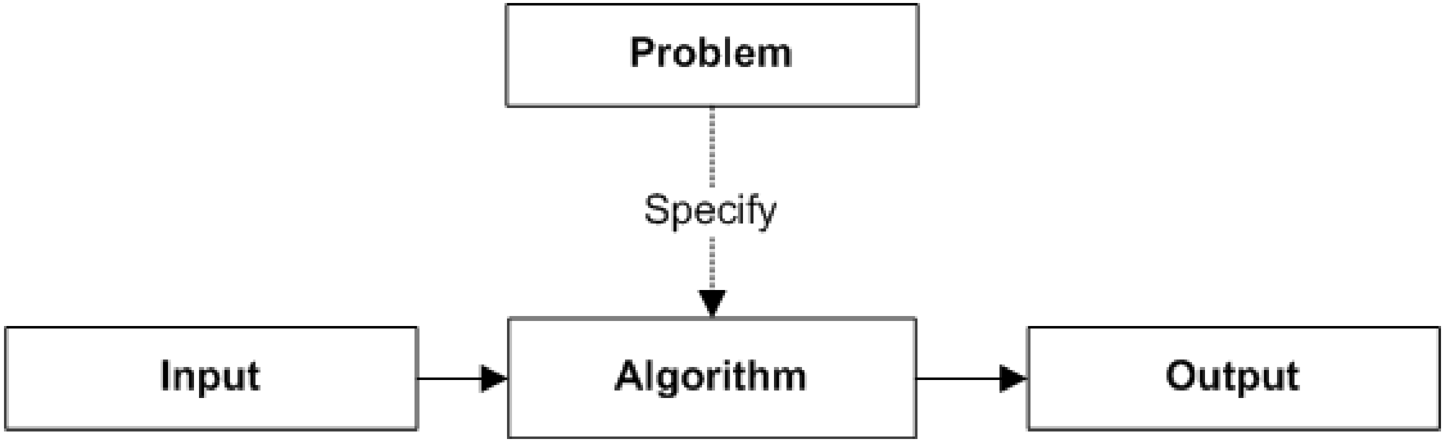
- **Try and specify a few examples**
  - Convert Miles to Kilometres
  - Find a patient record in a hospital database
  - Find a face in an image
  - Route planning




Gen 34 species 14 genome 14 (37%)
Fitness: -11 Max Fitness: 4322

What is an Algorithm? /4

- **Each algorithm should specify each of the following:**
  - Name and purpose
  - Input and output
  - Unambiguously specified, finite sequence of steps
  - Termination condition or terminating state



Algorithm Example /1

- **Algorithm: Selection Sort**
  - **Purpose**: Sorts elements in an unsorted array of integers in ascending (non-decreasing) order
  - **Input**: An array of integers, i[0]…i[n-1]
  - **Output**: A sorted array of integers in ascending order
  - **Steps**:

1. For each cur = 0 to n-2
   - Determine the minimum value from i[cur+1] … i[n-1]
   - Swap i[cur] value with the minimum value from i[cur] … i[n-1]
2. Output i[0]…i[n-1] and terminate

Algorithm Example /2

- **Algorithm: Selection Sort – Illustrative Scenario:**

i[0]	i[1]	i[2]	i[3]	i[4]	i[5]
15	35	5	64	36	11
5	**35**	15	64	36	11
5	**11**	15	**64**	36	35
5	11	15	**35**	36	64

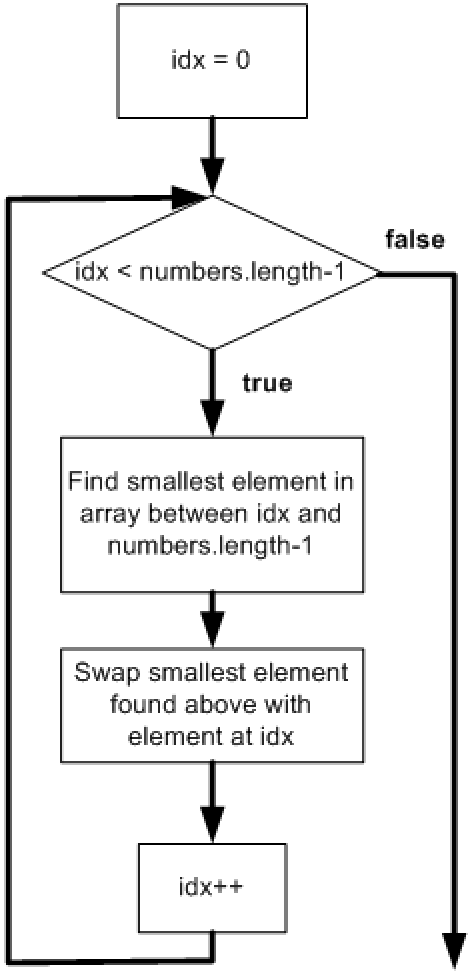
Algorithm Specification

- **Algorithms need to be specified clearly and unambiguously**
  - However, this goal does not always require coding

- **Algorithms can be specified using natural language or pseudocode**
  - The language used has to be structured and explicit with no assumptions or implications left open
  - Each step has to be clearly marked, and termination conditions clearly identified

- **Algorithm: Convert Canadian Dollars to Yen**
  - Step 1. Obtain CAD-to-JPY conversion rate from the inputted conversion table
  - Step 2. Multiply the inputted value in Canadian Dollars with the CAD-to-JPY conversion rate
  - Step 3. Output the multiplication result in Yen, and terminate

Algorithm Specification


idx = 0
idx < numbers.length-1
false
true
Find smallest element in array between idx and numbers.length-1
Swap smallest element found above with element at idx
idx++

What is a Data Structure?

- **Algorithms operate on various data items**
  - These items are typically organized in a manner that is conducive to storage and manipulation

- **Data Structure**
  - A coherent organization of related data items for efficient storage and usage
  - Allows for an organized way to manage large amounts of data in an efficient manner
  - Facilitates design and use of algorithms that meet efficiency parameters

- **In C++, we can use structs or classes for data structures**

Structure Types in C++

- **Structs are typically declared globally**
  - But no memory is allocated at declaration time

- **Example:**

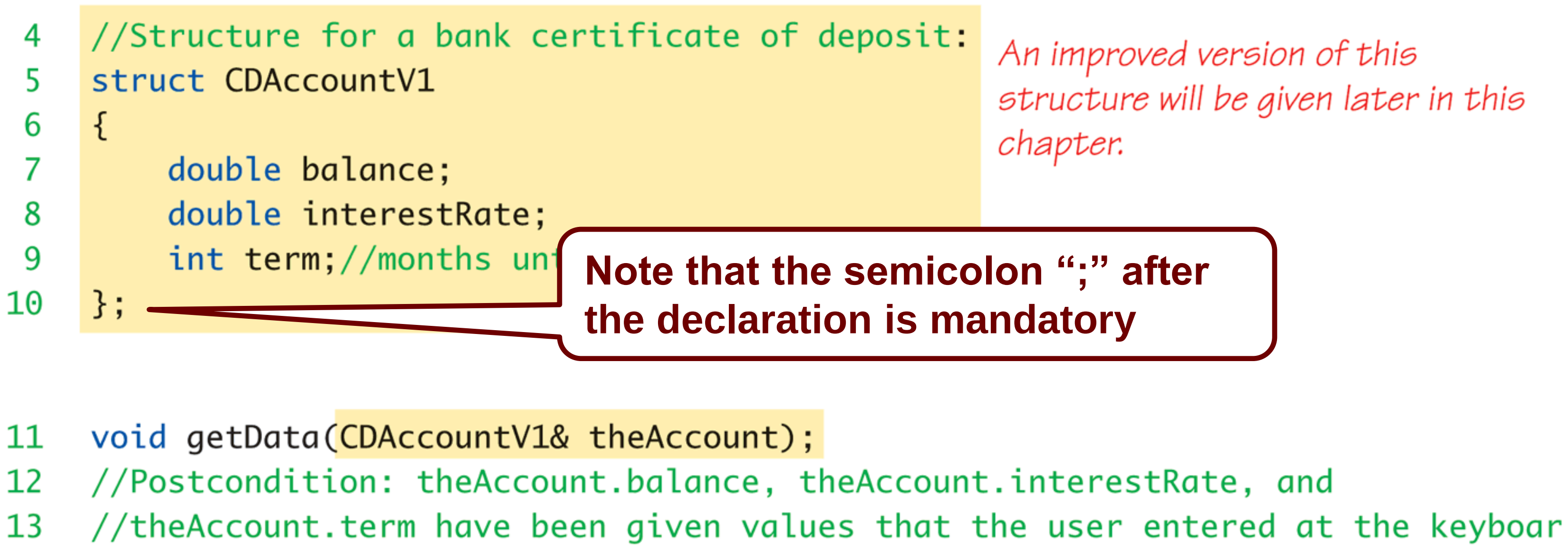
  -
    ```
    struct CDAccountV1   // Name of the new struct type
    {
            double balance;   // member names
            double interestRate;
            int term;
    };
    ```

Note that the semicolon “;” after the declaration is mandatory

- **With structure type defined, we can now declare variables of this new type and allocate memory:**
  - *CDAccountV1 account;*

Accessing Structure Members /1

- **Dot Operator is used to access members**
  - *account.balance*
  - *account.interestRate*
  - *account.term*
- **These are called the member variables**
  - The parts of the structure variable



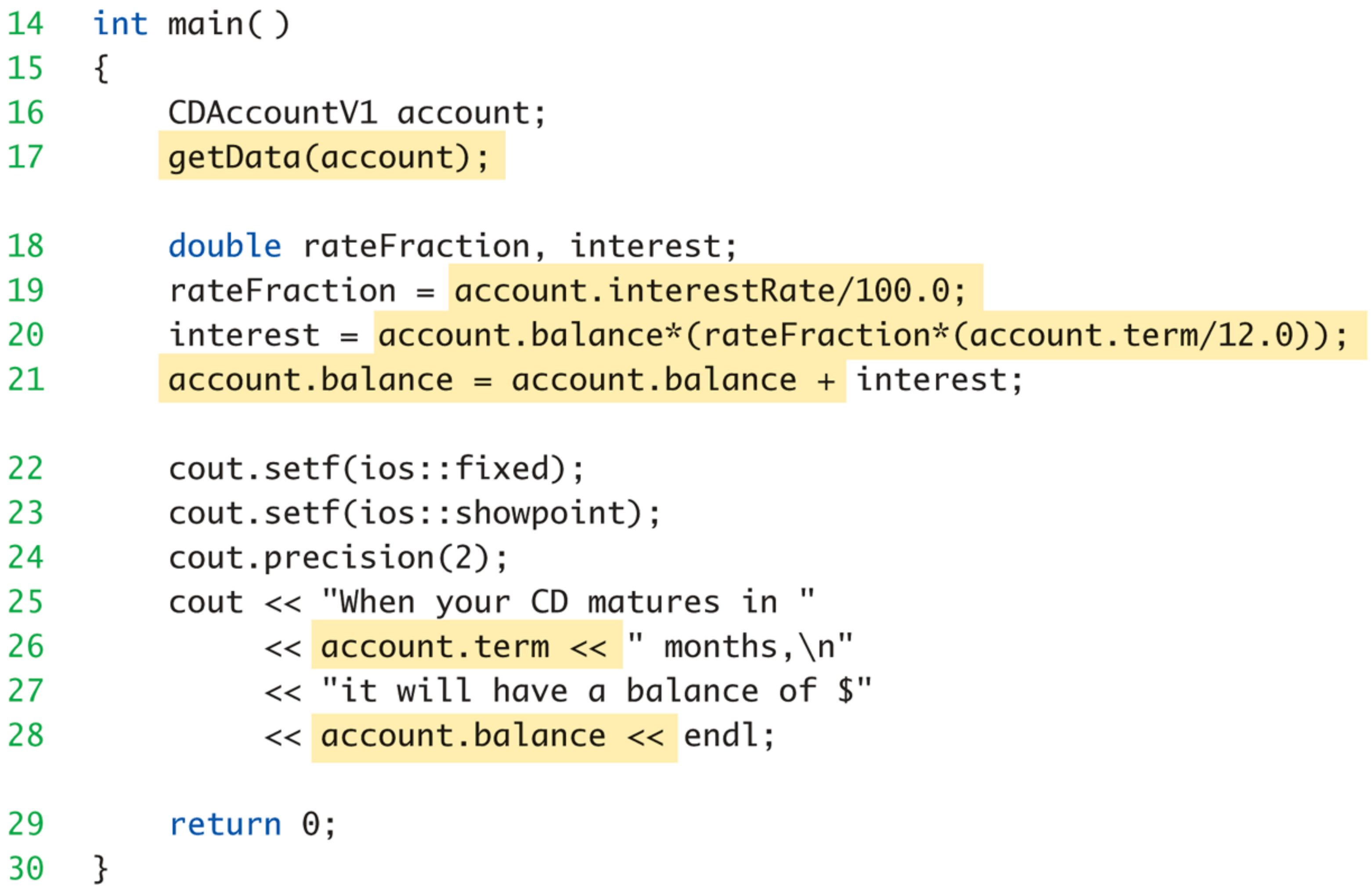
```
//Structure for a bank certificate of deposit:
struct CDAccountV1
{
    double balance;
    double interestRate;
    int term;//months un
};
```

An improved version of this structure will be given later in this chapter.

Note that the semicolon “;” after the declaration is mandatory

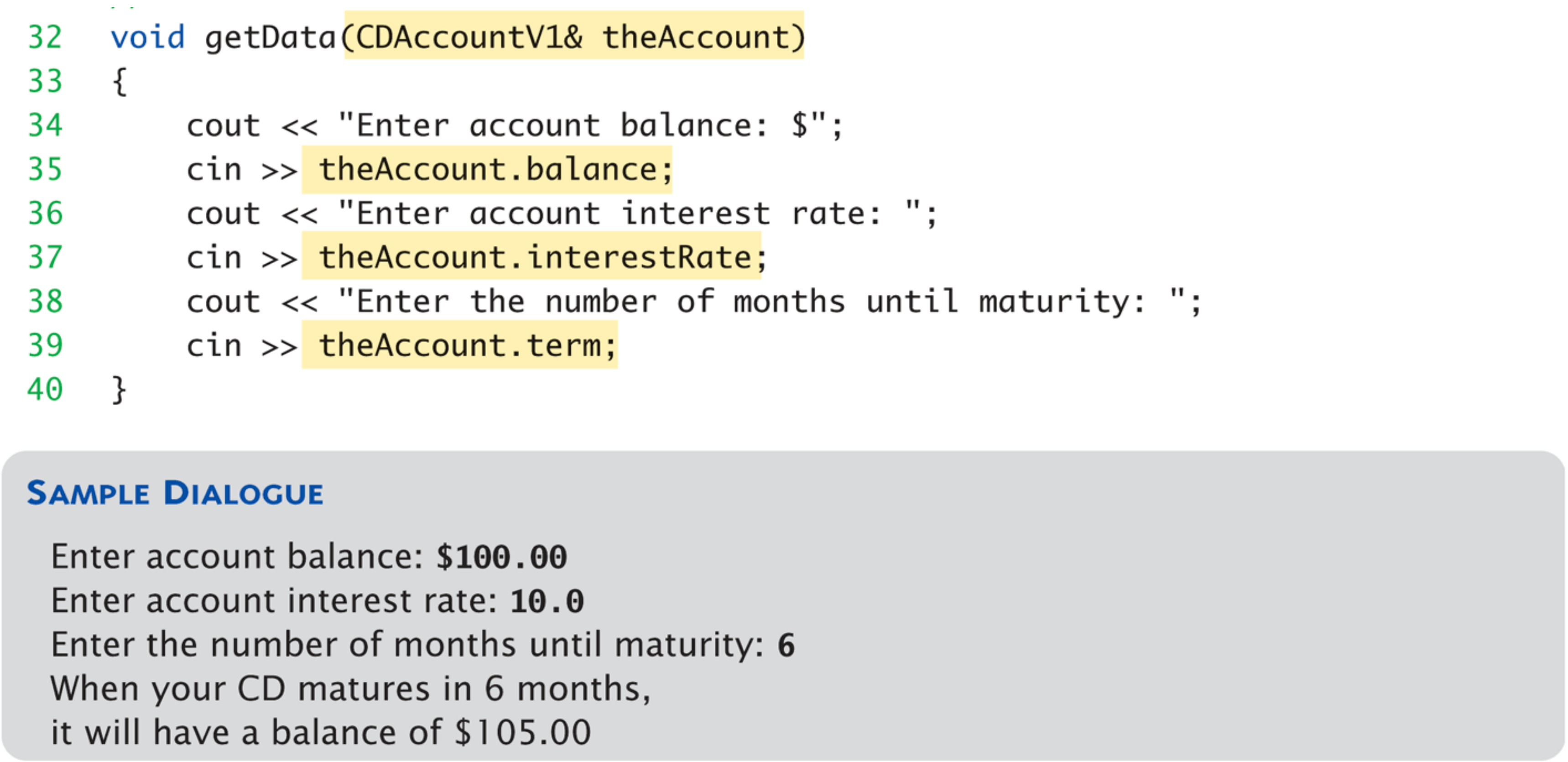
```
void getData(CDAccountV1& theAccount);
//Postcondition: theAccount.balance, theAccount.interestRate, and
//theAccount.term have been given values that the user entered at the keyboar
```

Accessing Structure Members /2



```
int main( )
{
    CDAccountV1 account;
    getData(account);

    double rateFraction, interest;
    rateFraction = account.interestRate/100.0;
    interest = account.balance*(rateFraction*(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;
}
```

Accessing Structure Members /3



```
void getData(CDAccountV1& theAccount)
{
    cout << "Enter account balance: $";
    cin >> theAccount.balance;
    cout << "Enter account interest rate: ";
    cin >> theAccount.interestRate;
    cout << "Enter the number of months until maturity: ";
    cin >> theAccount.term;
}
```

Sample Dialogue

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

Structure Assignments

- **Given structure named CropYield**
  - Declare two structure variables:
    CropYield apples, oranges;
  - Both are variables of struct type CropYield
  - Simple assignments are legal: apples = oranges;
  - **This assignment is not copying the address but instead copying each member variable from oranges into apples**

- **Structs can also be returned by function**
  - Return type is the structure type
  - Return statement in function definition sends the structure variable back to the caller
  - Example: *CDAccountV1 getAccount();*

Classes

- **Focus of classes is on objects**
  - Integral concept for object-oriented programming

- **Object: Contains attributes and methods**
  - In C++, variables of the class type are objects

- **Example:**

```
class DayOfYear   // name of new class type
{
public:
          void output();       // member method
          int month;          // member attribute
          int day;
};
```

  - **Note that only the method declaration is provided**

Declaration vs definition??

Declaring Objects

- **Declared same as all variables**
  - Predefined types or structure types

- **Example:**
  - DayOfYear today, birthday;
  - Declares two objects of the class type DayOfYear (but no values have been set yet)

- **Objects include:**
  - **Attributes** – members are month and day
  - **Methods (member functions)** – members are output()

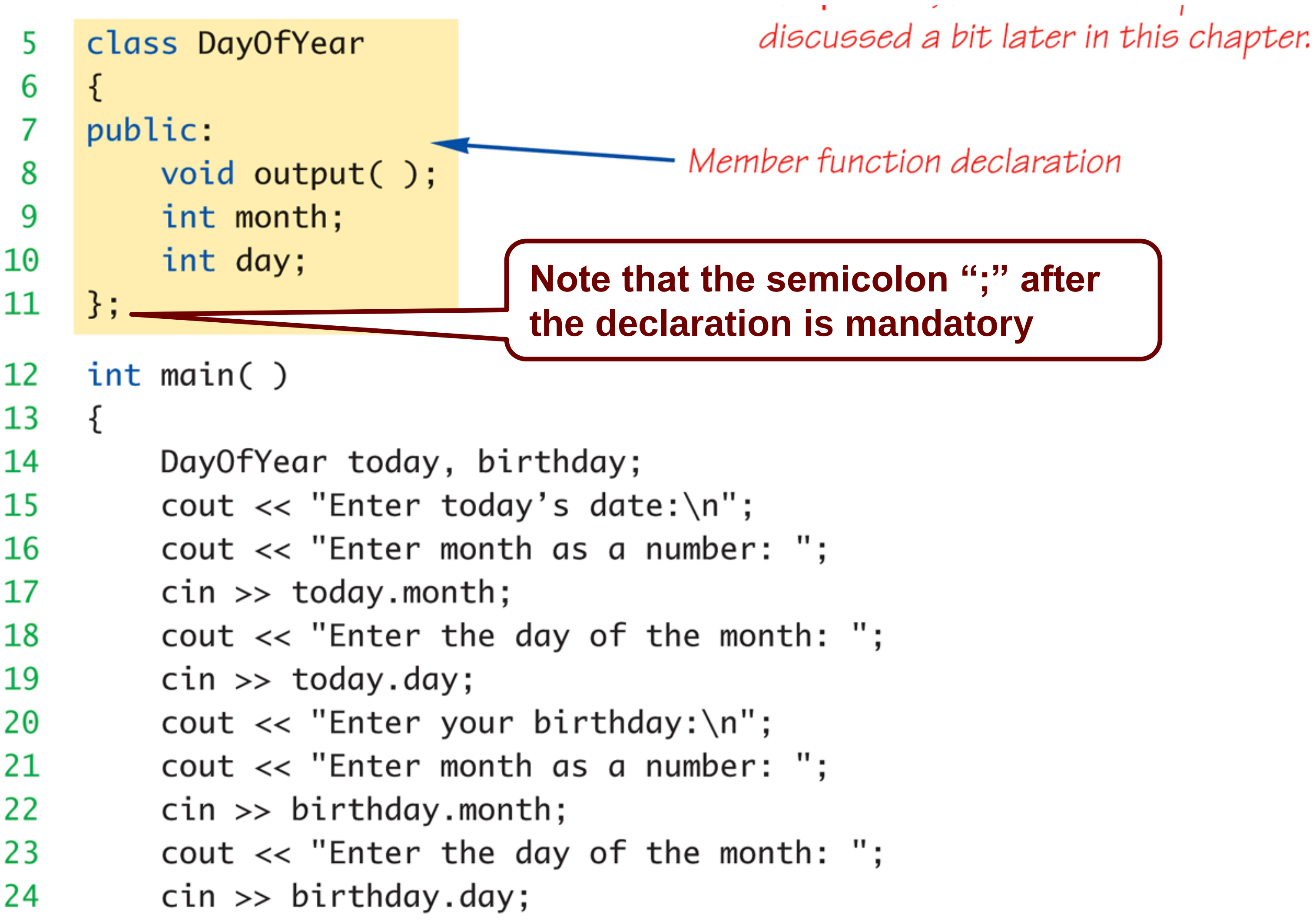
Class Member Access

- **Members are accessed using the dot operator**

- **Example:**
  - *today.month*
    today.day
  - And to access member function:
    today.output(); // Invokes member function

- **Must define or implement class member functions**
  - Like other function definitions, can be after main()
  - Must specify class:
    void DayOfYear::output() {…}
  - **:: is the scope resolution operator**
  - The item before :: is called the type qualifier
    - **Rule of thumb: "*a::b* means *b* is a member of *a*"**

Class With a Member Function /1



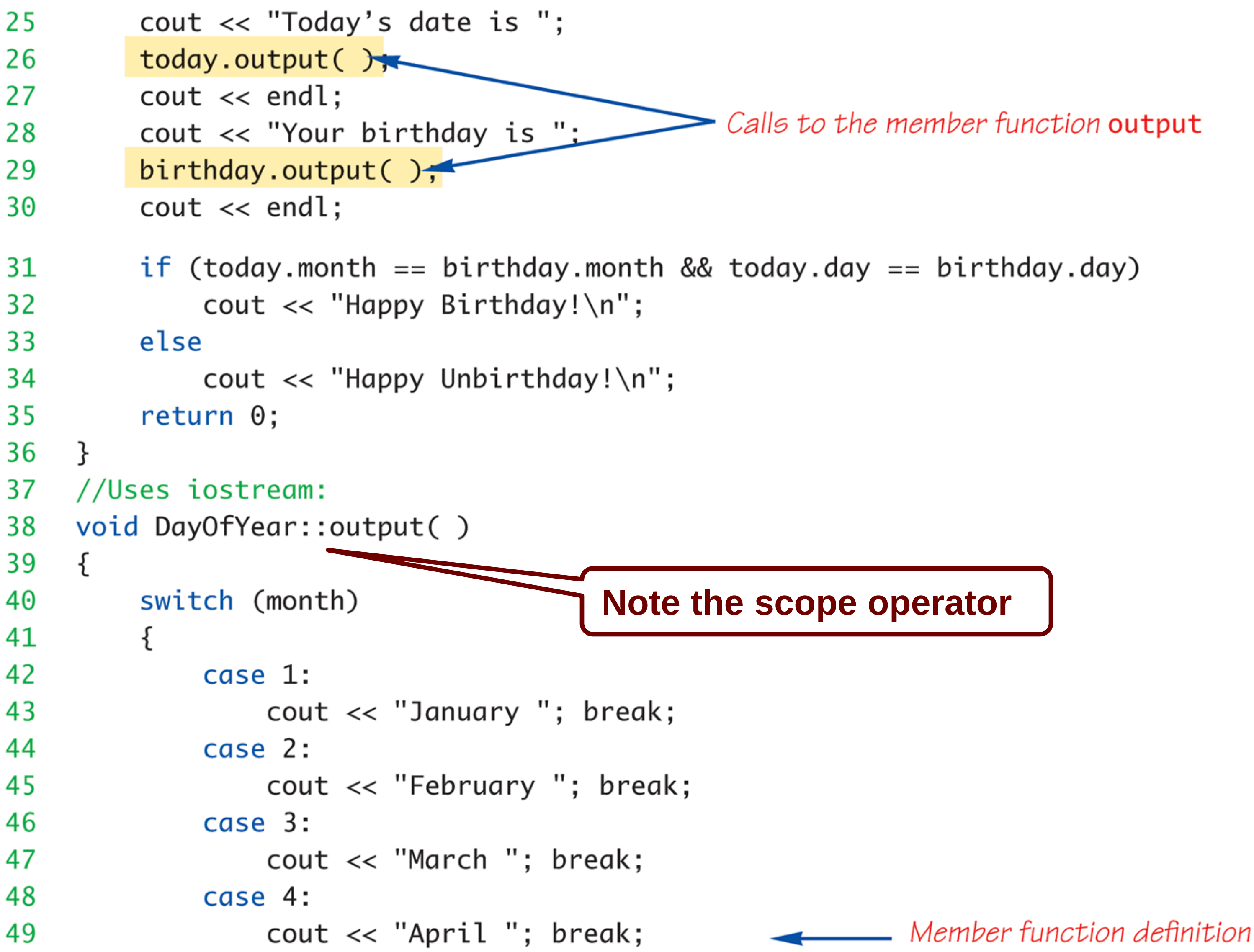
discussed a bit later in this chapter.

```
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;
```

Member function declaration

Note that the semicolon ";" after the declaration is mandatory

Class With a Member Function /2



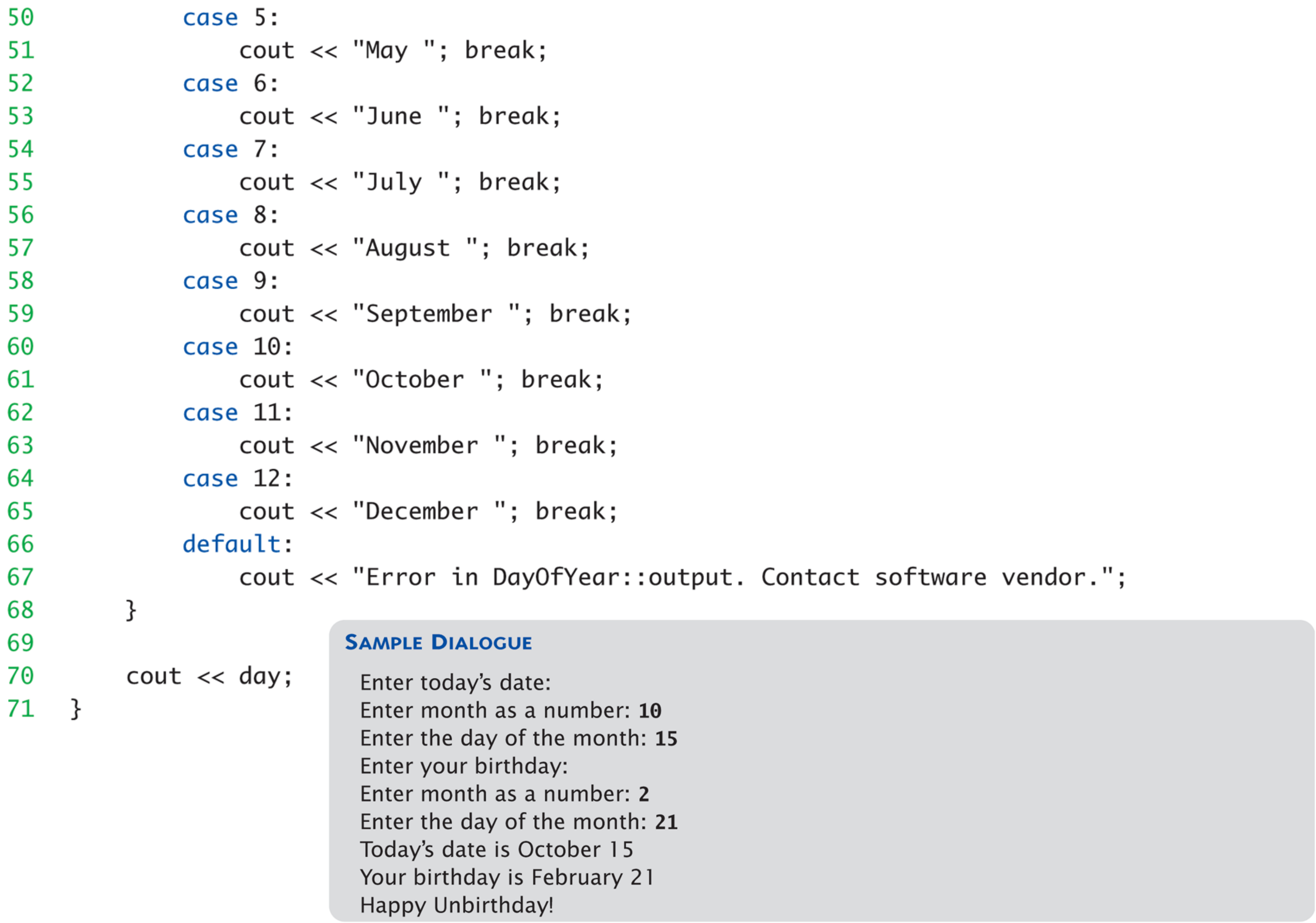
```
    cout << "Today's date is ";
    today.output( );
    cout << endl;
    cout << "Your birthday is ";
    birthday.output( );
    cout << endl;

    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( )
{
    switch (month)
    {
        case 1:
            cout << "January "; break;
        case 2:
            cout << "February "; break;
        case 3:
            cout << "March "; break;
        case 4:
            cout << "April "; break;
```

Calls to the member function **output**

Note the scope operator

Member function definition

Class With a Member Function /3



```
    case 5:
      cout << "May "; break;
    case 6:
      cout << "June "; break;
    case 7:
      cout << "July "; break;
    case 8:
      cout << "August "; break;
    case 9:
      cout << "September "; break;
    case 10:
      cout << "October "; break;
    case 11:
      cout << "November "; break;
    case 12:
      cout << "December "; break;
    default:
      cout << "Error in DayOfYear::output. Contact software vendor.";
  }

  cout << day;
}
```

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 October 15
Your birthday is February 21
Happy Unbirthday!
```

Dot and Scope Resolution Operator

- **Dot “.” Operator:**
  - Specifies member of particular object

- **Scope Resolution “::” Operator:**
  - Specifies what class the function definition comes from

- **Class is a full-fledged type**
  - Just like the built-in data types int, double, etc.
  - Can use class type like any other type
  - **Variables of a class type are simply called "objects“**

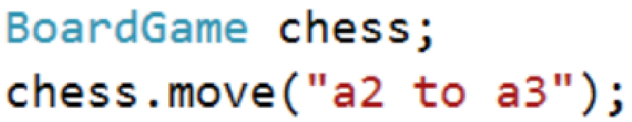
- **Can have parameters of a class type**
  - Can also use Pass by Value and Pass by Reference

Abstract Data Types

- **Abstract Data Type (ADT)**
  - A collection of data items given a name, purpose, and a set of operations that operate on the data items
  - With the ADT, only the interface (the functions) are exposed externally, and data organization is hidden

- **ADTs are often language independent**
  - We will implement ADTs in C++ with classes
  - C++ class defines the ADT

- **Example**



```
BoardGame chess;
chess.move("a2 to a3");
```

How does it keep track of the pieces?
Do we care?

Abstract Data Types

- **Encapsulation**
  - Class contains all the resources needed to function
  - It contains the attributes and methods (member functions) that operate on said attributes
  - Data not accessed directly, but rather through method calls
  - Class controls data manipulation, and is hidden from programmer

OOP Principles

- **Information Hiding**
  - Details of how methods are implemented within the class are not known to the user of the class
  - Only the interface is exposed (i.e., public function declarations and associated comments)

- **Data Abstraction**
  - Details of how data is manipulated within the class are not known to the user of the class

- **Two fundamental challenges in OO development are:**
  - Identifying classes/objects
  - Decomposing the system into classes/objects

Public and Private Members

- **Data in a class should almost always be designated as private**
  - Upholds principles of OOP, namely the data abstraction
  - Private member data preserves the internal object state
  - Allow data manipulation only via member functions
- **Public items (usually member functions) are accessible by the class users**
  - If there is no visibility declaration, private is default
  - Another visibility type, called protected, reserved for inheritance hierarchies (more on this later)

Public and Private Example /1

- **Example:**

  -
    ```
    class DayOfYear
    {
    public:
              void input();
              void output();
    private:
              int month;
              int day;
    };
    ```
  - Data members in the above example are private
  - Outside of the class definition code, other objects and functions have no direct access

Public and Private Example /2

- **Based on the previous example, declare object:**
  - *DayOfYear today;*

- **The object today can only access public members**
  - *cin >> today.month; // NOT ALLOWED!*
  - *cout << today.day;   // NOT ALLOWED!*

- **Must instead call public methods**
  - *today.input();*
  - *today.output();*

Accessor ("getter") and Mutator ("setter") Functions

- **Object needs to perform functions on its data**

- **Call accessor member functions to read data**
  - Also called "get member functions"
  - Simple retrieval of member data
  - Example: int getMonth();
int getDate();

- **Call mutator member functions to change data**
  - Manipulated based on the specific use case
  - Example: void setMonth(int newmonth);
void setDate(int newdate);

const Functions

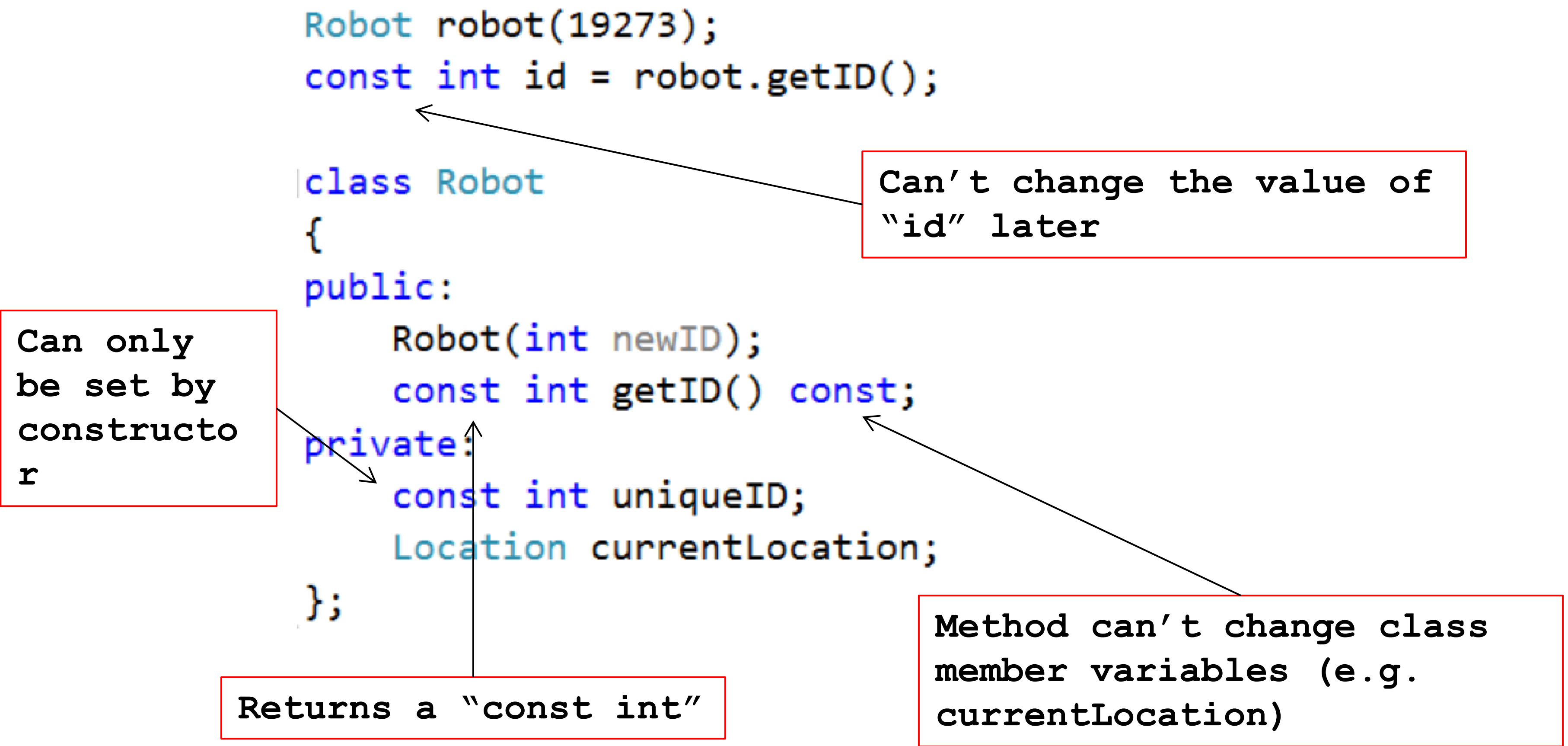
- **When to make function const?**
  - Constant functions not allowed to alter member data
  - Constant objects can only call constant member functions

- **Good style dictates:**
  - Any method that will not modify data should be made const

- **Use keyword *const* after function declaration**
  - *int Money::getCents() const*

const Trickery

- Depending on where it’s used, “const” can be tricky in C++, and is a source of massive debate among C++ gurus.



```
Robot robot(19273);
const int id = robot.getID();
```

```
class Robot
{
public:
    Robot(int newID);
    const int getID() const;
private:
    const int uniqueID;
    Location currentLocation;
};
```

Can’t change the value of “id” later

Can only be set by constructor

Returns a “const int”

Method can’t change class member variables (e.g. currentLocation)

Class Constructors /1

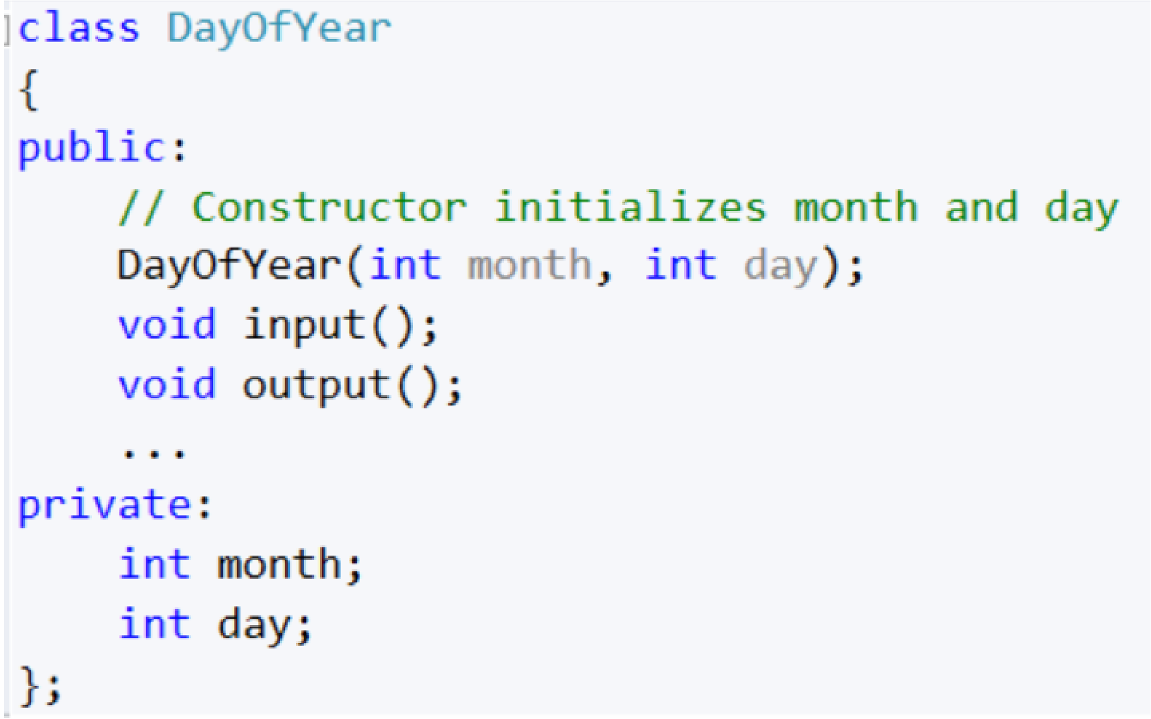
- **Used to initialize objects (class instances)**
  - Initialize some or all member variables
  - Other actions possible as well

- **A special kind of member function**
  - Automatically called when object is instantiated
  - One of the key building blocks of OOP

- **Constructors defined like any member function**
  - Must have the same name as the respective class
  - They cannot return a value, not even void

Class Constructors /2

- **Class definition with constructor:**

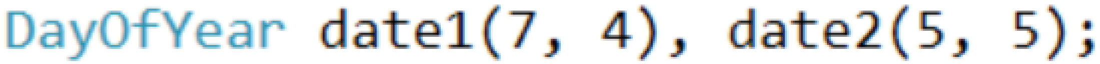


```
class DayOfYear
{
public:
    // Constructor initializes month and day
    DayOfYear(int month, int day);
    void input();
    void output();
    ...
private:
    int month;
    int day;
};
```

  - **Constructor is in the public section**
  - If private, could never instantiate objects

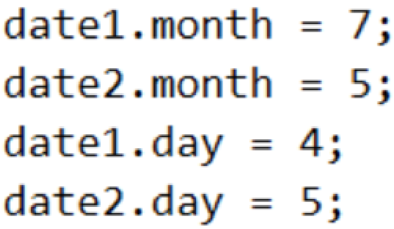
Calling Constructors

- **Instantiate objects:**



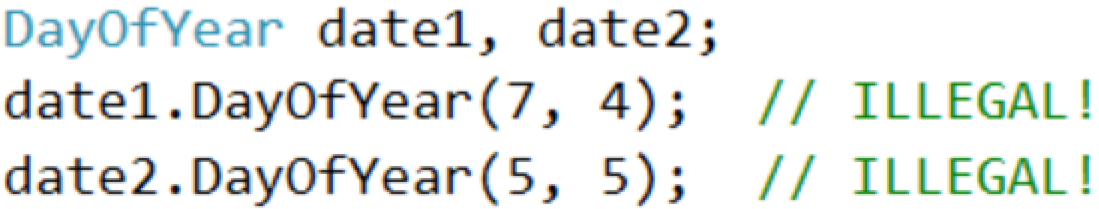
```
DayOfYear date1(7, 4), date2(5, 5);
```

- **Objects are created when the constructor is called**
  - Values in brackets passed as arguments to constructor
  - Member variables month, day initialized:



```
date1.month = 7;
date2.month = 5;
date1.day = 4;
date2.day = 5;
```

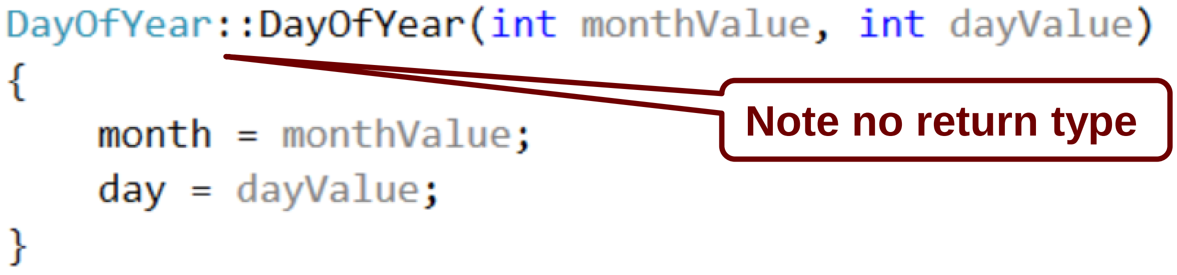
- **Consider:**



```
DayOfYear date1, date2;
date1.DayOfYear(7, 4);  // ILLEGAL!
date2.DayOfYear(5, 5);  // ILLEGAL!
```

Constructor Code

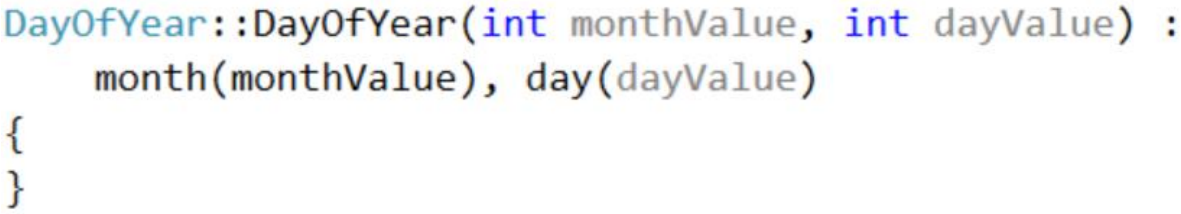
- **Constructors can be defined like other member functions:**



```
DayOfYear::DayOfYear(int monthValue, int dayValue)
{
    month = monthValue;
    day = dayValue;
}
```

Note no return type

- **Previous definition equivalent to:**



```
DayOfYear::DayOfYear(int monthValue, int dayValue) :
    month(monthValue), day(dayValue)
{
}
```

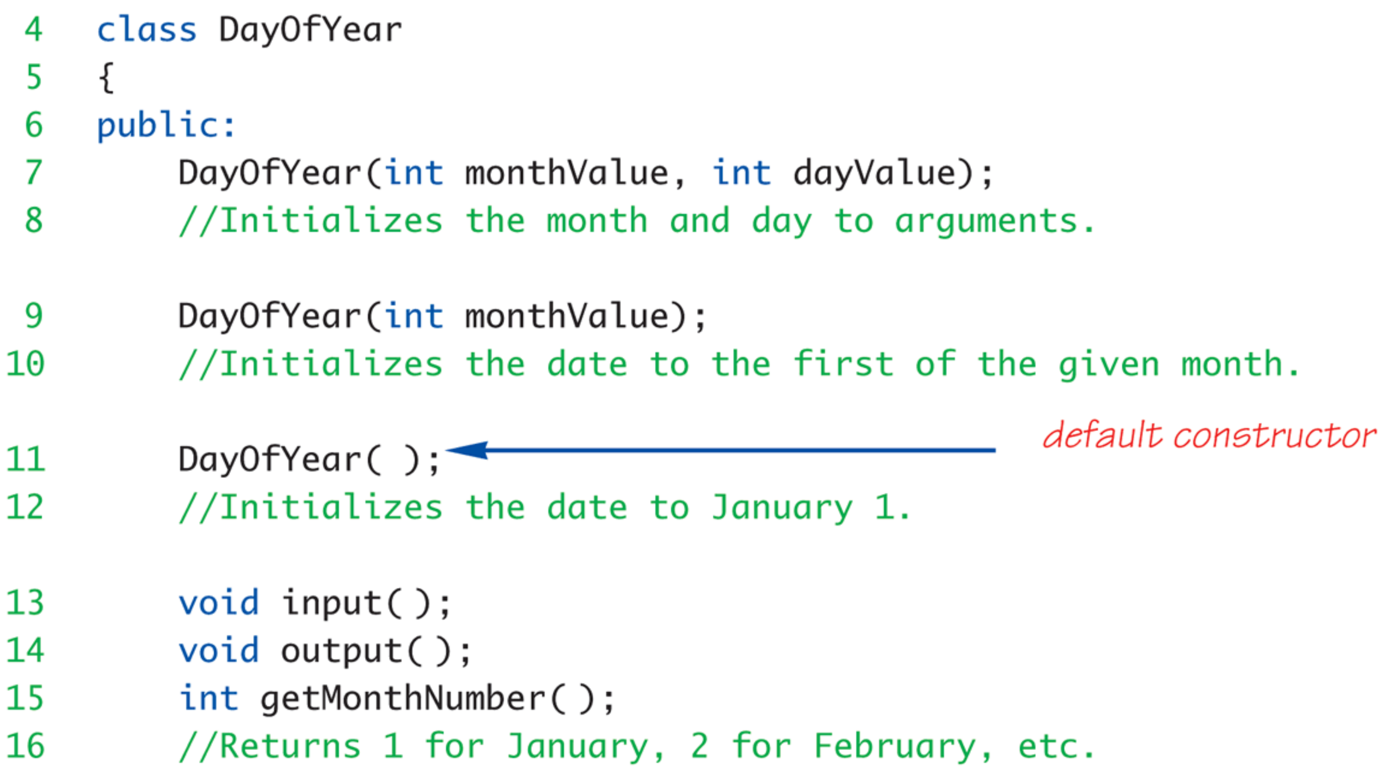
  - Third line (the initialization section) is left empty
  - **This definition is a more preferred style**

Constructor Additional Purpose

- **Constructor body does not have to be empty**
  - Use it to validate the entered data
  - **Ensure that only the appropriate data is assigned to class private member variables**
  - Very useful OOP recommendation

- **Can overload constructors just like other functions**
  - Provide constructors for all viable argument lists
  - Particularly for different number of arguments
  - **Recall that each constructor definition requires a different constructor signature/declaration**

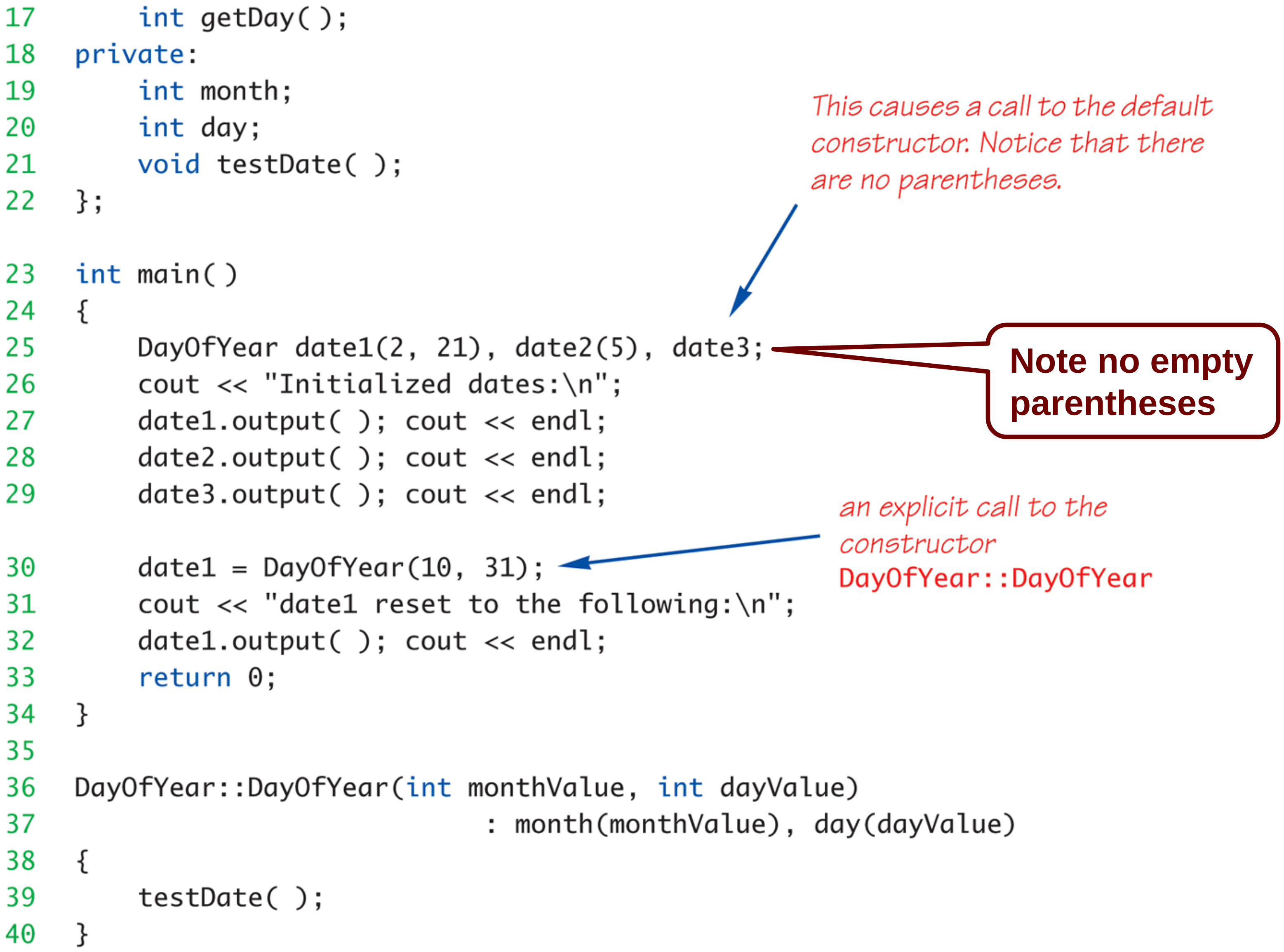
Class with Constructors Example /1



```
class DayOfYear
{
public:
    DayOfYear(int monthValue, int dayValue);
    //Initializes the month and day to arguments.

    DayOfYear(int monthValue);
    //Initializes the date to the first of the given month.

    DayOfYear( );
    //Initializes the date to January 1.

    void input( );
    void output( );
    int getMonthNumber( );
    //Returns 1 for January, 2 for February, etc.
```

default constructor

Class with Constructors Example /2



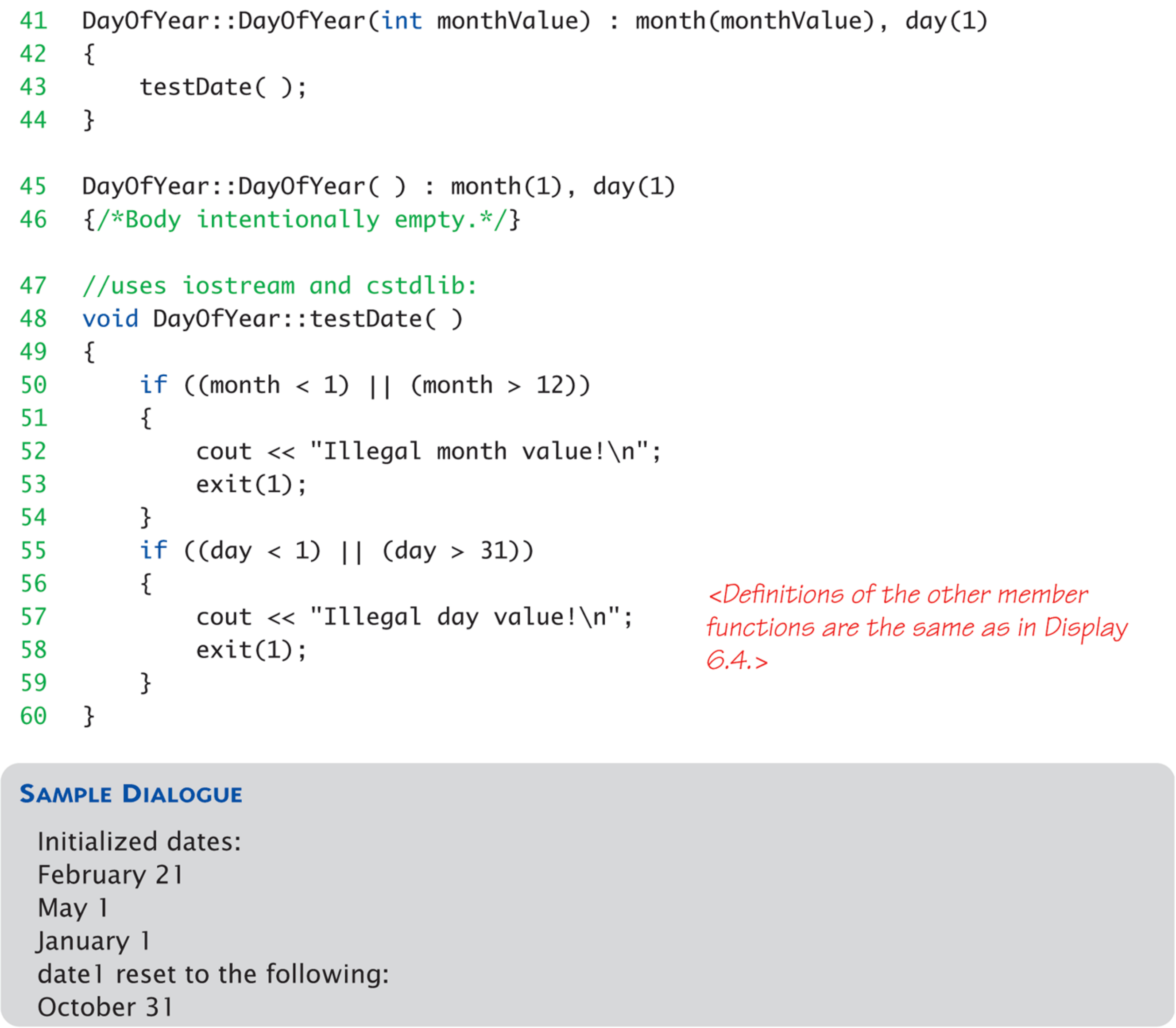
```
    int getDay( );
private:
    int month;
    int day;
    void testDate( );
};

int main( )
{
    DayOfYear date1(2, 21), date2(5), date3;
    cout << "Initialized dates:\n";
    date1.output( ); cout << endl;
    date2.output( ); cout << endl;
    date3.output( ); cout << endl;

    date1 = DayOfYear(10, 31);
    cout << "date1 reset to the following:\n";
    date1.output( ); cout << endl;
    return 0;
}

DayOfYear::DayOfYear(int monthValue, int dayValue)
                          : month(monthValue), day(dayValue)
{
    testDate( );
}
```

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

Note no empty parentheses

an explicit call to the constructor `DayOfYear::DayOfYear`

Class with Constructors Example /3



```
DayOfYear::DayOfYear(int monthValue) : month(monthValue), day(1)
{
    testDate( );
}

DayOfYear::DayOfYear( ) : month(1), day(1)
{/*Body intentionally empty.*/}

//uses iostream and cstdlib:
void DayOfYear::testDate( )
{
    if ((month < 1) || (month > 12))
    {
        cout << "Illegal month value!\n";
        exit(1);
    }
    if ((day < 1) || (day > 31))
    {
        cout << "Illegal day value!\n";
        exit(1);
    }
}
```

<Definitions of the other member functions are the same as in Display 6.4.>

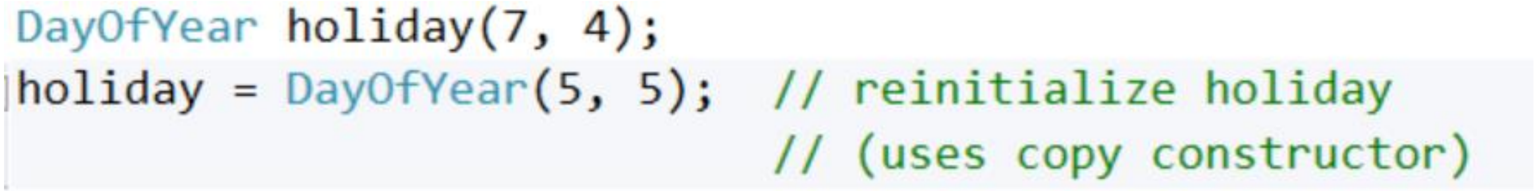
SAMPLE DIALOGUE

```
Initialized dates:
February 21
May 1
January 1
date1 reset to the following:
October 31
```

Explicit Constructor Calls

- **Class constructor can be called again after the object has already been initialized**
  - Such a call returns anonymous object which can then be assigned to a local instance
  - This is a convenient method of setting member variables

- **Example:**



```
DayOfYear holiday(7, 4);
holiday = DayOfYear(5, 5);   // reinitialize holiday
                             // (uses copy constructor)
```

  - Explicit constructor call returns new anonymous object
  - Assigned back to the current object

Default Constructor

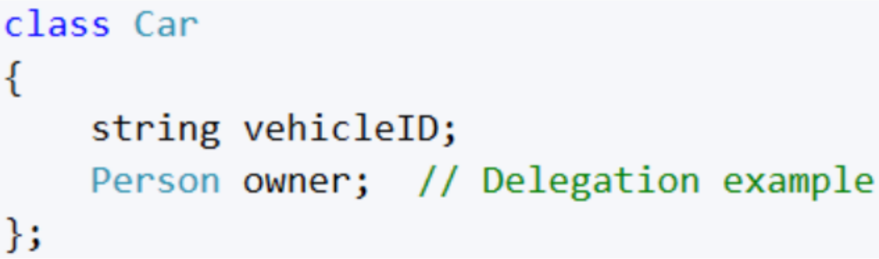
- **Defined as a constructor with no arguments**
  - One should always be defined
  - **If no constructors are defined then one is automatically generated by the compiler**
  - If any constructor is defined then no default constructor (with no arguments) is automatically created
  - **Rule of Thumb: ALWAYS DEFINE AT LEAST ONE CONSTRUCTOR AND THE DESTRUCTOR** (more on this later)

- **If no default constructor is declared or generated**
  - Then one cannot declare:
    MyClass myObject;

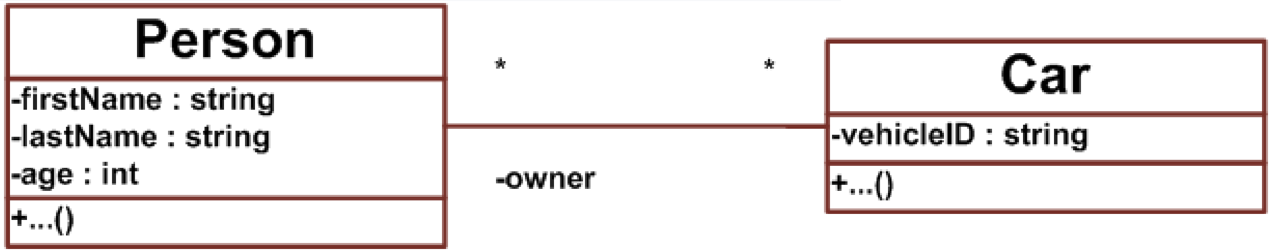
Class Type Member Variables

- **Class member variables can be of any type**
  - Including objects of other classes

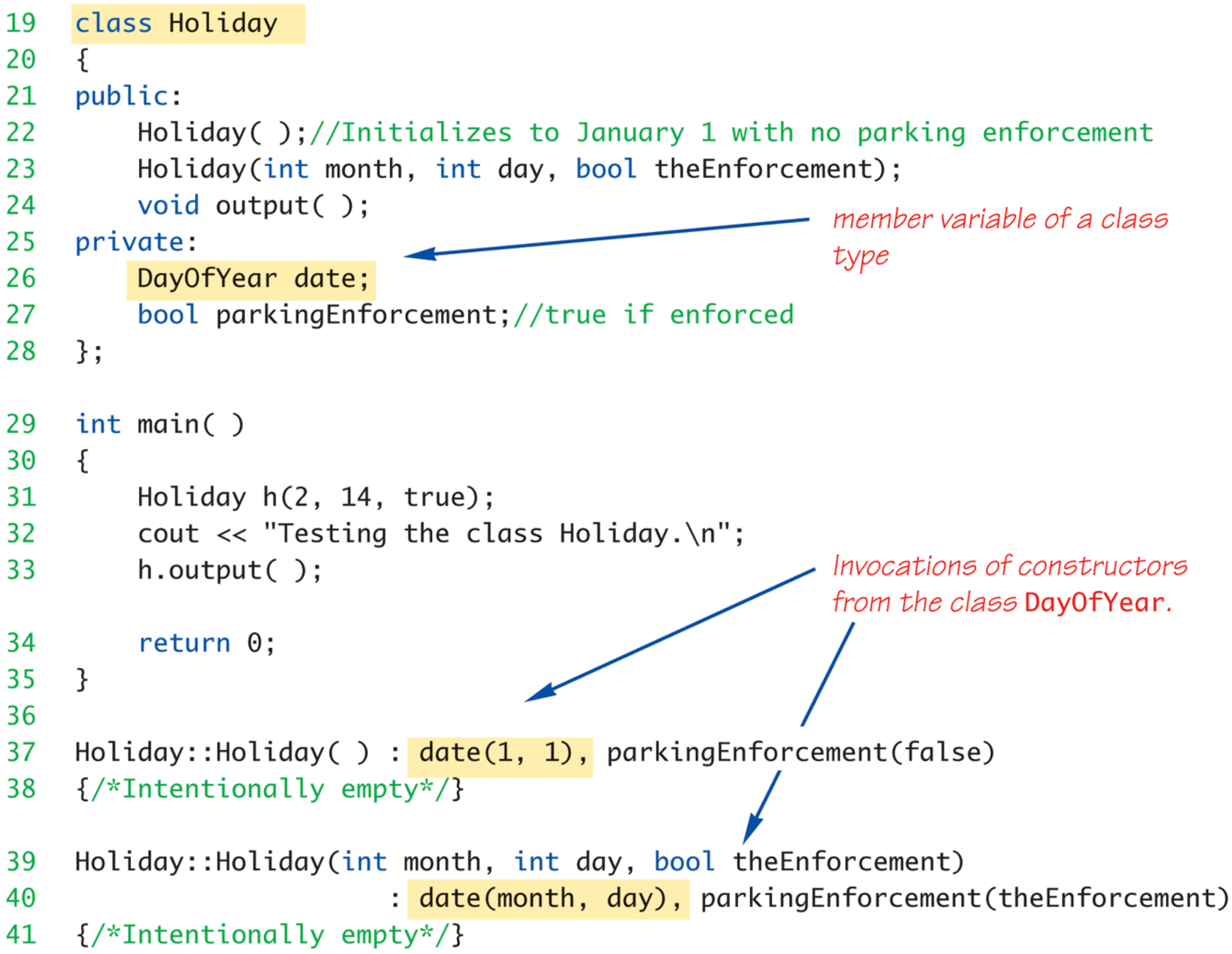
- **Delegation:**
  - Objects of class A include objects (one or many) of class B as member variables
  - Important OOP principle
- **Example:**



```
class Car
{
    string vehicleID;
    Person owner;  // Delegation example
};
```



Class Member Variable Example /1

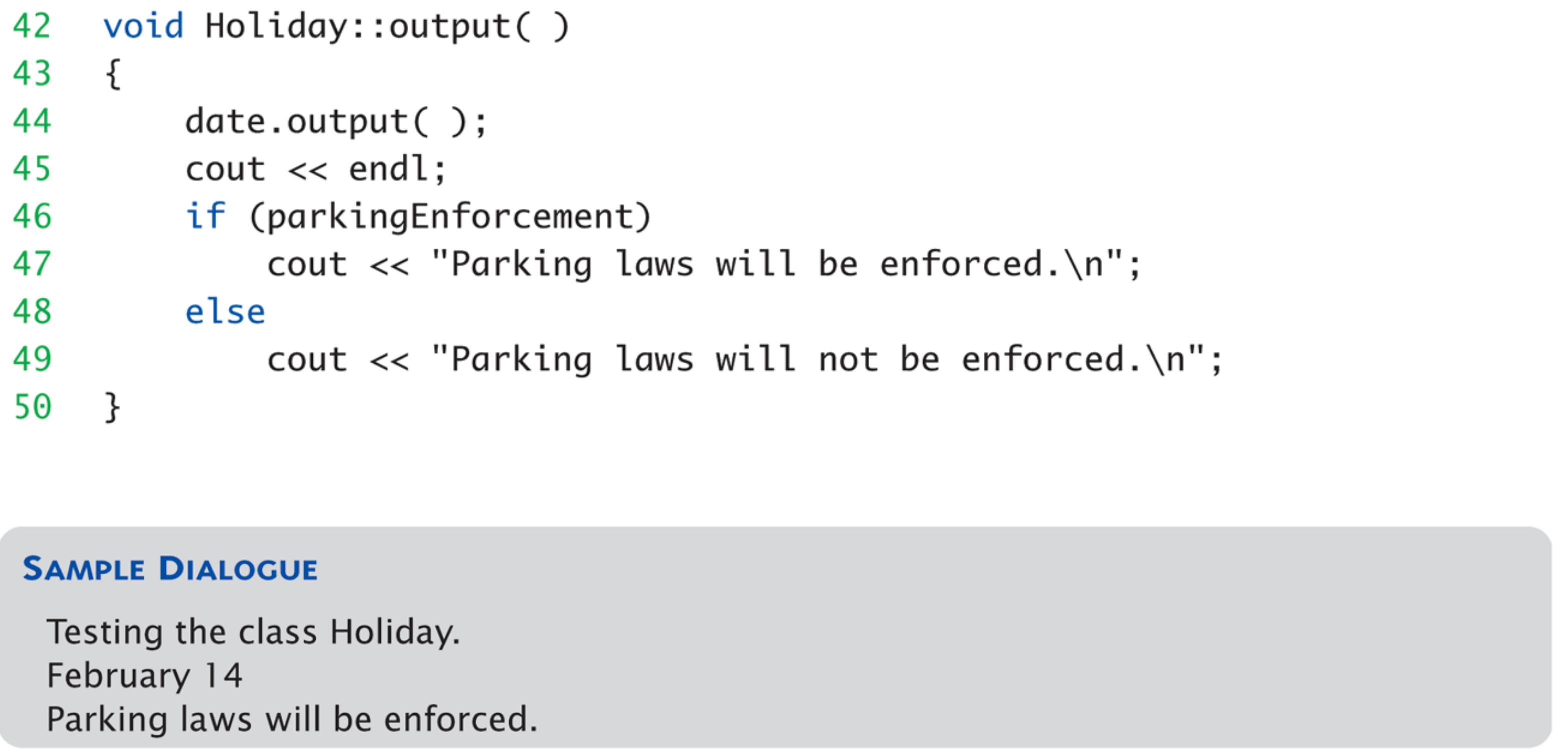


```
class Holiday
{
public:
    Holiday( );//Initializes to January 1 with no parking enforcement
    Holiday(int month, int day, bool theEnforcement);
    void output( );
private:
    DayOfYear date;
    bool parkingEnforcement;//true if enforced
};

int main( )
{
    Holiday h(2, 14, true);
    cout << "Testing the class Holiday.\n";
    h.output( );

    return 0;
}

Holiday::Holiday( ) : date(1, 1), parkingEnforcement(false)
{/*Intentionally empty*/}

Holiday::Holiday(int month, int day, bool theEnforcement)
                    : date(month, day), parkingEnforcement(theEnforcement)
{/*Intentionally empty*/}
```

member variable of a class type

Invocations of constructors from the class DayOfYear*.*

Class Member Variable Example /2



```
void Holiday::output( )
{
    date.output( );
    cout << endl;
    if (parkingEnforcement)
        cout << "Parking laws will be enforced.\n";
    else
        cout << "Parking laws will not be enforced.\n";
}
```

Sample Dialogue

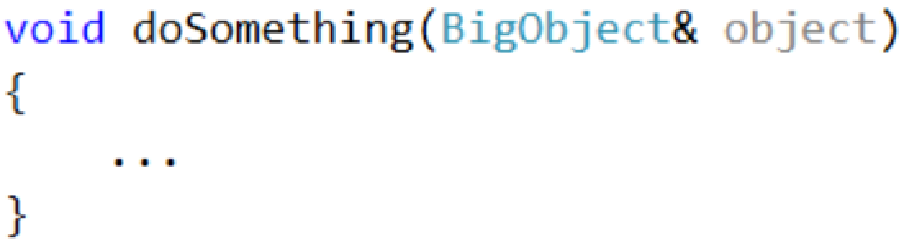
```
Testing the class Holiday.
February 14
Parking laws will be enforced.
```

Passing Classes as Parameters

- **For large data types such as classes:**
  - It is desirable to use **Pass by Reference** mechanism

Why?

  - Even if the functions will not make modifications



```
void doSomething(BigObject& object)
{
        ...
}
```

- **To protect the class argument**
  - Place the keyword **const** before the class type
  - Attempt to modify the parameter results in compiler error
  - Note that the approach is all or nothing: protects both member attributes and methods

Destructors

- **Performs the opposite function of a constructor**
  - Called when the object’s scope is closed to deallocate the memory assigned to the object
  - Or when the dynamically allocated object is explicitly deleted (more on this later)
  - Never call using ~Destructor; this is only used for declaration

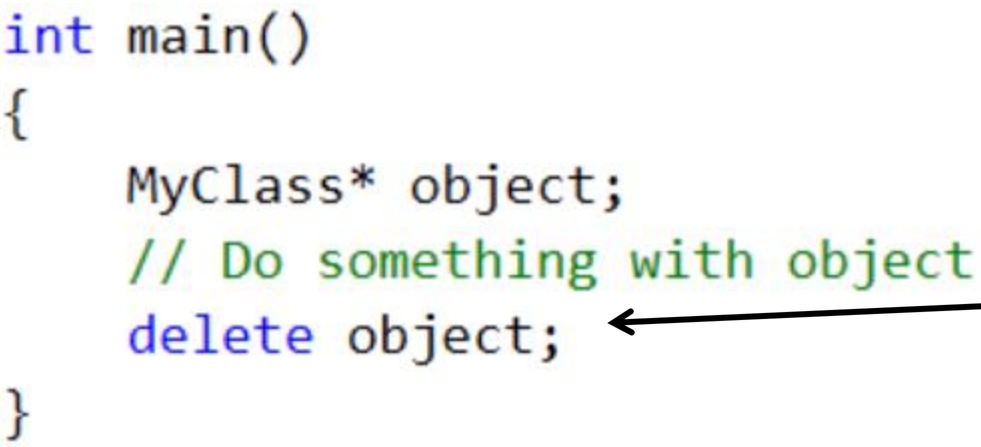
- **Destructor must be named the same as the class**
  - Just with a ~ (tilde) sign preceding its name
  - Example:
    Server(); // Constructor

    ~Server(); // Destructor

- **Important Rule: Each class has only one destructor**

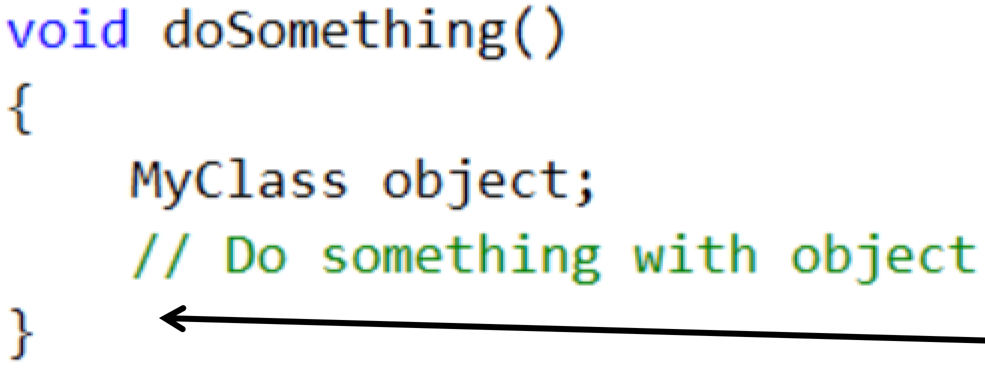
When is the Destructor Called?

(more on pointers next class)



```
int main()
{
    MyClass* object;
    // Do something with object
    delete object;
}
```

Explicitly delete object.
Destructor called.



```
void doSomething()
{
    MyClass object;
    // Do something with object
}
```

Class is created on the stack.
“object” goes out of scope.
Destructor implicitly called.

Objectives

Core Content:

- Introduction to Algorithms and Data Structures
- Defining C++ Classes
- Public and Private Class Members
- Accessor and Mutator Functions
- Class Constructors and Destructors

Additional Information:

- **Copy Constructors**
- **Using Inline Functions and Static Member Data**
- **Operator Overloading**
- **Declaring const Functions**
- **Declaring friend Functions**

Lecture Notes Summary

- **What do you need to know?**
  - What is an algorithm (p4)
  - Algorithm vs. process (p5)
  - Algorithm components (p7)
  - Algorithm specification (p10)
  - What is a data structure (p11)
  - Structs in C++ (p12)
  - Accessing struct members (p14)
  - Struct assignment (p16)
  - Classes in C++ (p17)
  - Declaring objects (p18)
  - Class member access (p19)
  - Class with a member function (p20)
  - Dot and scope operators (p23)
  - What is an abstract data type (ADT) (p24)
  - Information hiding and data abstraction (p25)
  - Public and private class members (p26)
  - Accessor and mutator member functions (p29)
  - Const member functions (p30)
  - Class constructors (p31)
  - Calling constructors (p33)
  - Defining constructors (p34)
  - Additional use of constructors (p35)
  - Default constructor (p40)
  - What is delegation (p41)
  - Const before class type (p44)
  - Class destructor (p45)

Food for Thought

- **Read:**
  - Chapter 1 (Introduction) from the course handbook

- **Additional Readings:**
  - Chapter 2 from “Data Structures and Other Objects Using C++” by Main and Savitch
  - Review Chapters 6, 7, 8 from “Absolute C++” by Savitch and Mock
    - Review the material discussed above in more detail

Copy Constructor

- **Special kind of a constructor**
  - Provided to make copies of an existing class
  - A default copy constructor is provided by the compiler

- **Typical signatures:**
  - *Money(const Money& copyme); // provided by the compiler*
  - *Money(Money& copymetoo);*

- **Correct definition:**
  - *Money::Money(const Money& copyme): a1(copyme.a1)… {}*

- **Incorrect signatures:**
  - *Money(Money* notcorrect); // not a copy constructor*
  - *Money(Money invalidcopy); // infinite loop*

Static Members /1

- **Static Member Variables**
  - Place keyword **static** before type
  - All objects of class share one copy of the variable
  - If one object changes it then all objects see the change

- **Useful for tracking objects**
  - How often a member function is called?
  - How many objects exist at given time?

- **Singleton Design Pattern**
  - Ensures only one instance of a class
  - Based on a static instance of a class

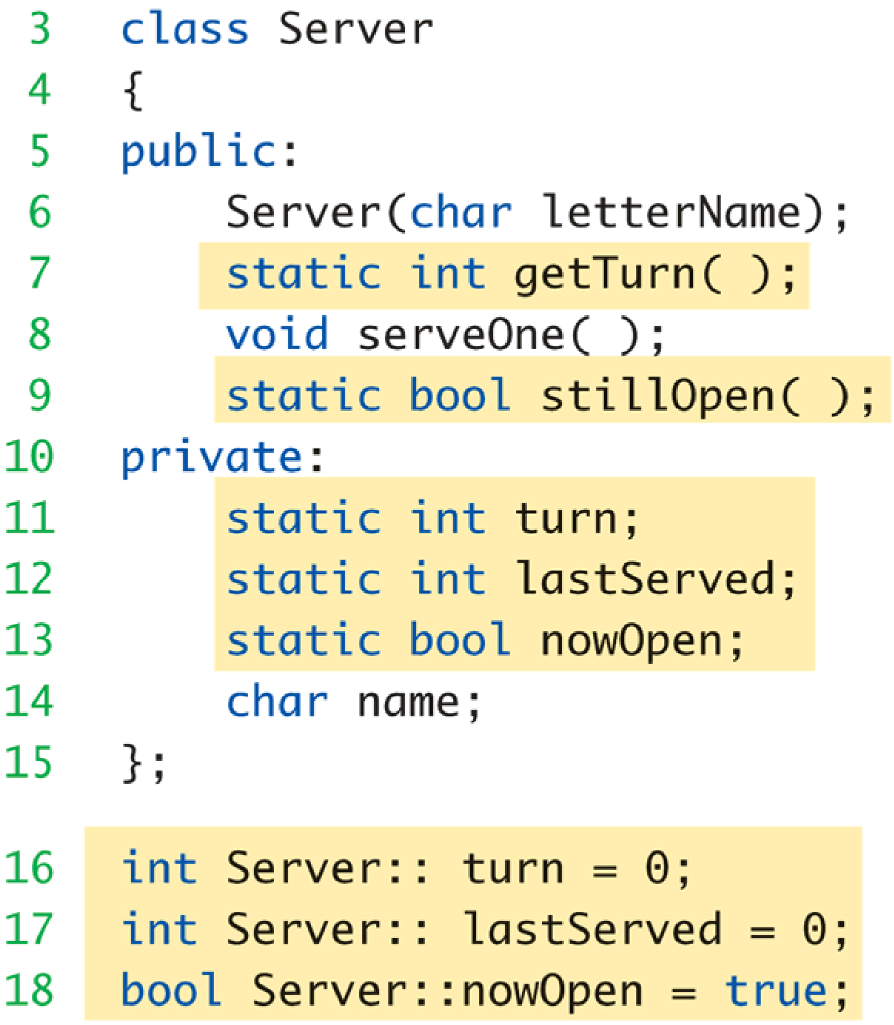
Static Members /2

- **Member functions can be static**
  - If no access to object data is needed, we can make the function static
  - It still must be a member of the class

- **The static function can then be called outside class**
  - Using the :: operator as
    Server::getTurn();
  - Or from class objects as
    myObject.getTurn();

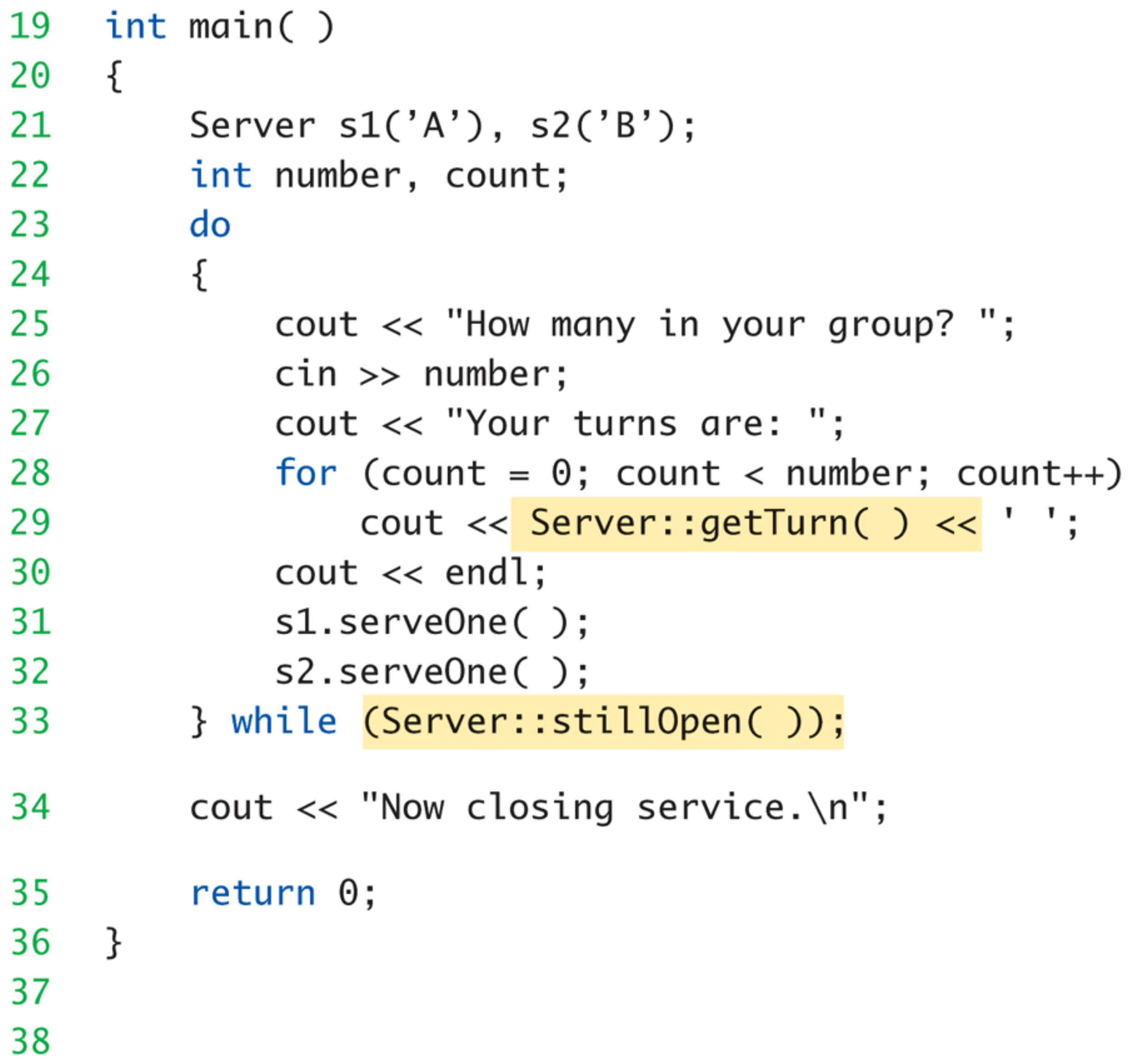
- **Key limitation:**
  - Can only use static data and functions

Static Members Example /1



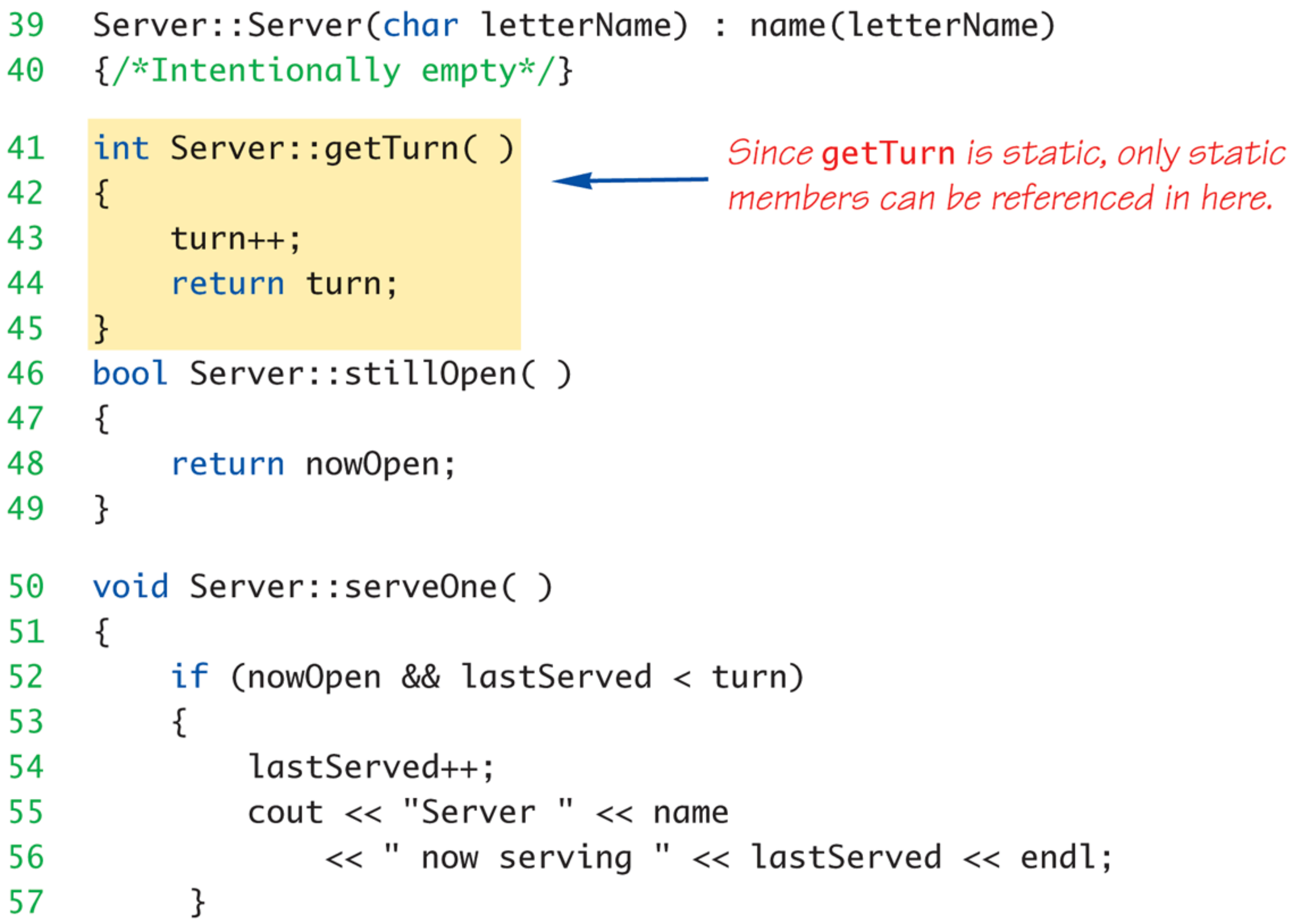
```
class Server
{
public:
    Server(char letterName);
    static int getTurn( );
    void serveOne( );
    static bool stillOpen( );
private:
    static int turn;
    static int lastServed;
    static bool nowOpen;
    char name;
};

int Server:: turn = 0;
int Server:: lastServed = 0;
bool Server::nowOpen = true;
```

Static Members Example /2



```
int main( )
{
    Server s1(’A’), s2(’B’);
    int number, count;
    do
    {
        cout << "How many in your group? ";
        cin >> number;
        cout << "Your turns are: ";
        for (count = 0; count < number; count++)
            cout << Server::getTurn( ) << ' ';
        cout << endl;
        s1.serveOne( );
        s2.serveOne( );
    } while (Server::stillOpen( ));
    cout << "Now closing service.\n";
    return 0;
}
```

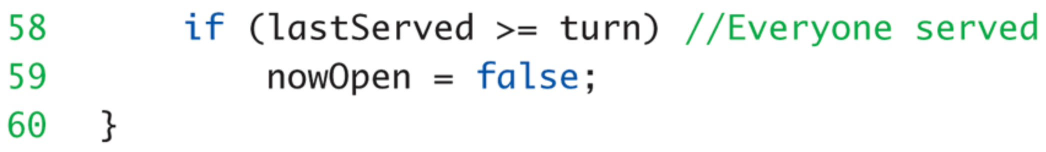
Static Members Example /3



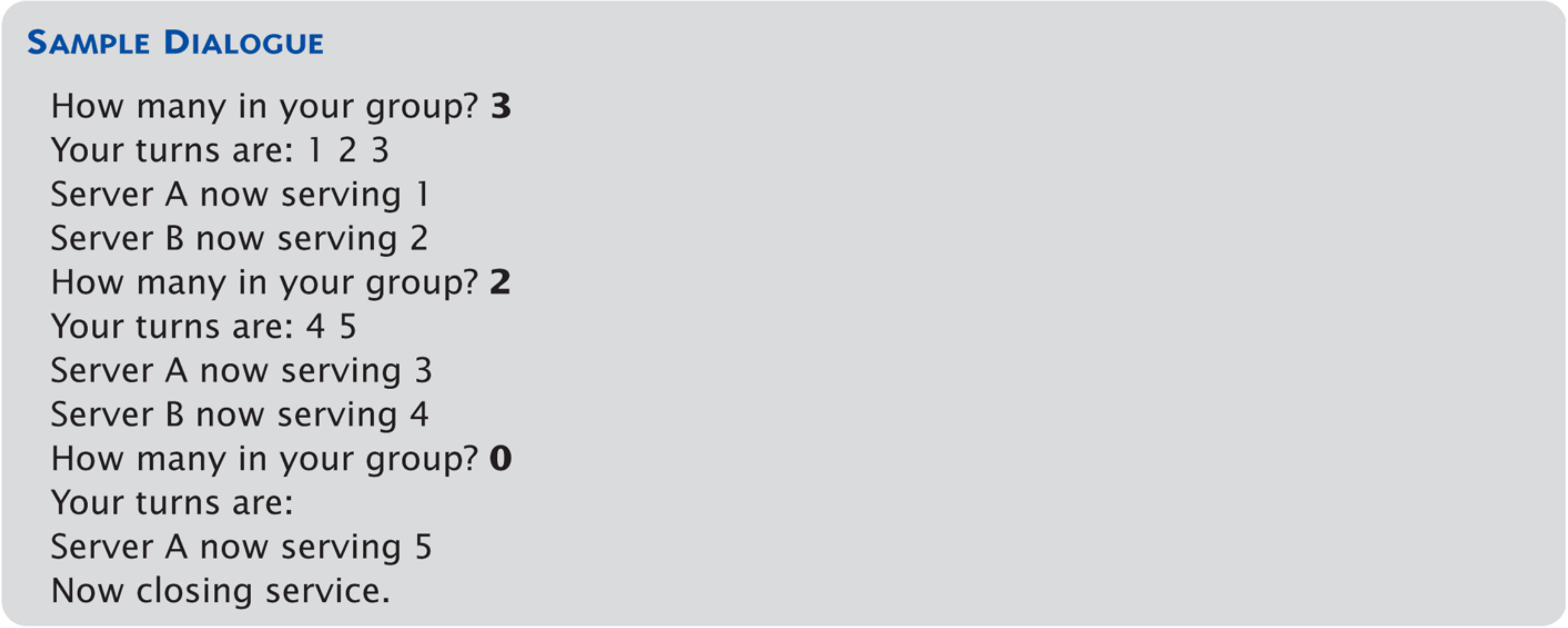
```
Server::Server(char letterName) : name(letterName)
{/*Intentionally empty*/}

int Server::getTurn( )
{
    turn++;
    return turn;
}
bool Server::stillOpen( )
{
    return nowOpen;
}

void Server::serveOne( )
{
    if (nowOpen && lastServed < turn)
    {
        lastServed++;
        cout << "Server " << name
            << " now serving " << lastServed << endl;
     }
```

Since **getTurn** *is static, only static members can be referenced in here.*

Static Members Example /4



```
    if (lastServed >= turn) //Everyone served
        nowOpen = false;
}
```



Sample Dialogue

```
How many in your group? 3
Your turns are: 1 2 3
Server A now serving 1
Server B now serving 2
How many in your group? 2
Your turns are: 4 5
Server A now serving 3
Server B now serving 4
How many in your group? 0
Your turns are:
Server A now serving 5
Now closing service.
```

Inline Functions

- **Use the keyword inline before function declaration**
  - Use for very short functions only
  - Code actually inserted in place of call
  - Eliminates calling overhead
  - **If used for longer functions, can lead to creation of large compilation units and thereby becoming inefficient**

- **For non-member functions:**
  - Use **inline** in function declaration and function heading

- **For class member functions:**
  - Place function definition (function implementation) in the class declaration; also called implicit inlining
  - Can also declare a function in the class declaration and then later define it separately as an inline function

Operator Overloading Introduction /1

- **Operators +, -, %, ==, etc are really just functions**
  - Just called with different syntax: x + 7
  - "+" is a binary operator with x & 7 as operands
  - Think of it as: +(x, 7)
  - "+" is the function name
  - x & 7 are the arguments
  - Function "+" returns the sum of its arguments

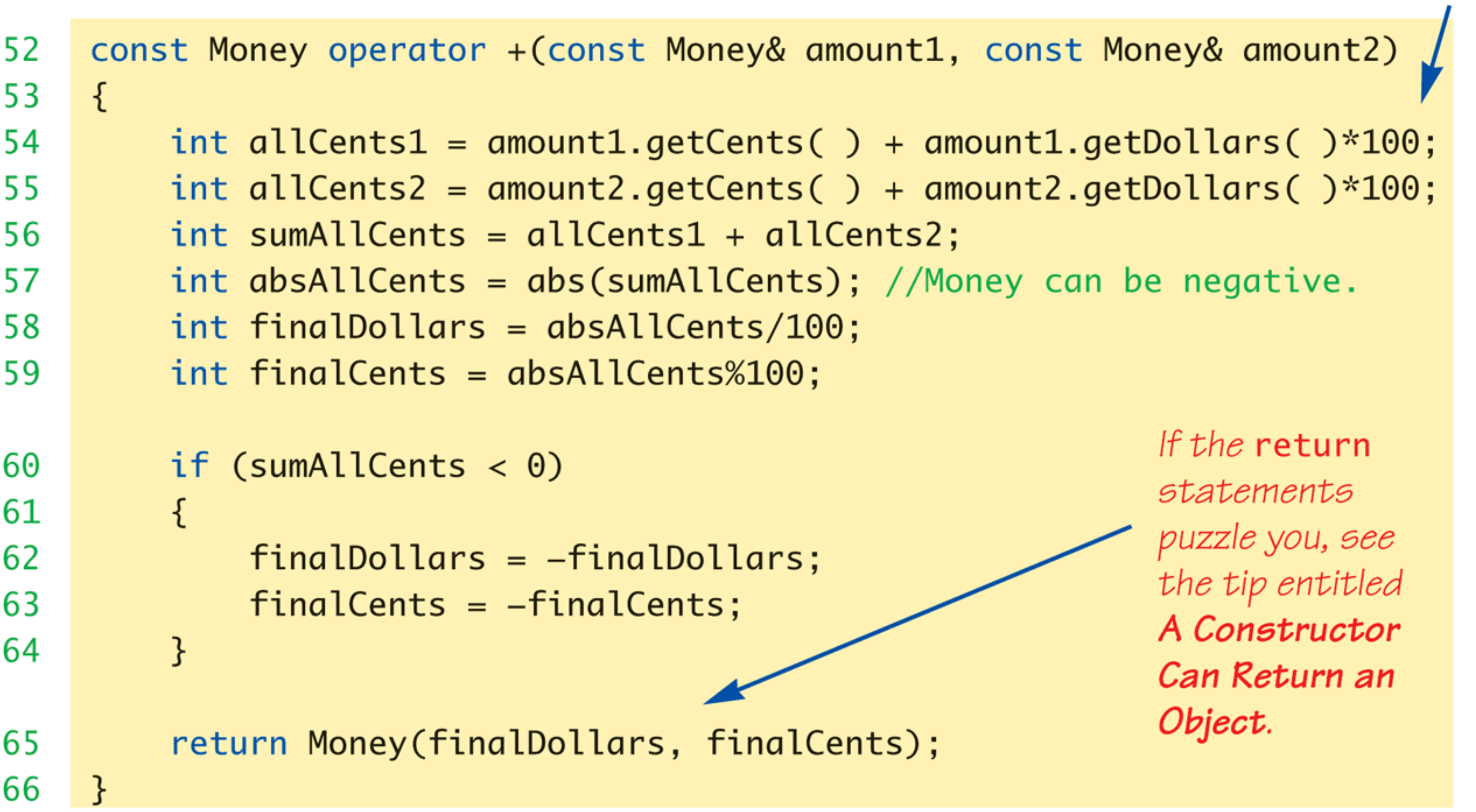
Operator Overloading Introduction /2

- **Built-in operators**
  - Such as, +, -, = , %, ==, /, *
  - Already work for built-in C++ types

- **These can be overloaded to handle custom types**
  - Overloading operators is similar to overloading functions
  - Operator itself is the name of the function

- **Example Declaration:**
  - *const Money operator +(const Money& amount1,*
    const Money& amount2);
  - Overloads + for operands of type Money
  - Allows addition of objects of type Money
  - **Note that overloaded "+" is not a member function**

Overloaded "+" for the Money Type

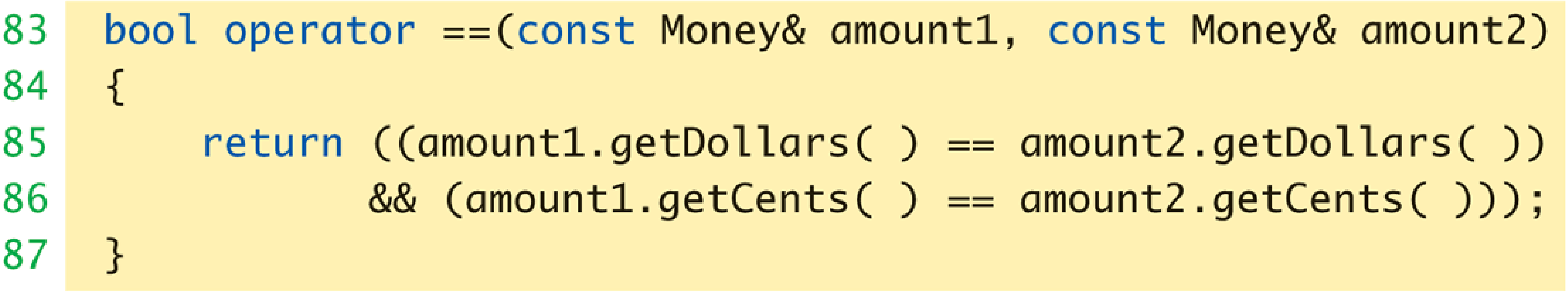


```
const Money operator +(const Money& amount1, const Money& amount2)
{
    int allCents1 = amount1.getCents( ) + amount1.getDollars( )*100;
    int allCents2 = amount2.getCents( ) + amount2.getDollars( )*100;
    int sumAllCents = allCents1 + allCents2;
    int absAllCents = abs(sumAllCents); //Money can be negative.
    int finalDollars = absAllCents/100;
    int finalCents = absAllCents%100;

    if (sumAllCents < 0)
    {
        finalDollars = -finalDollars;
        finalCents = -finalCents;
    }

    return Money(finalDollars, finalCents);
}
```

If the **return** *statements puzzle you, see the tip entitled* ***A Constructor Can Return an Object.***

Overloaded "=="

- **Overload the equality operator, ==**
  - Enables comparison of Money objects
  - Declaration:
    bool operator ==(const Money& amount1,
    const Money& amount2);
  - Returns bool type for true/false equality
  - Again, it’s a non-member function
    (like "+" overload)



```
bool operator ==(const Money& amount1, const Money& amount2)
{
    return ((amount1.getDollars( ) == amount2.getDollars( ))
           && (amount1.getCents( ) == amount2.getCents( )));
}
```

Returning by const Value

- **Recall return statement in "+" overload for Money type:**
  - return Money(finalDollars, finalCents);
  - Returns an invocation of Money class
  - **So this constructor actually returns an object, which is called an anonymous object**

- **Consider "+" operator overload again:**
  - *const Money operator +(const Money& amount1,*
    const Money& amount2);
  - Returns a constant object
  - **Why make the return object read only?**

Returning by non-const Value

- **Consider not using const in the following declaration:**
  - *Money operator +(const Money& amount1,*
    const Money& amount2);
- **Consider expression that calls: m1 + m2**
  - m1 & m2 are Money objects, and the object returned is also a Money object
  - We could therefore invoke member functions on object returned by expression m1+m2
  - (m1+m2).output(); // Not a problem: no modification
  - (m1+m2).input();   // Problem: modifies the return object
  - **Should not modify an anonymous object**
  - **So, we define the return type as const (read only)**

Overloading Unary Operators

- **C++ has specific unary operators:**
  - Defined as taking one operand
  - e.g., - (negation):
    x = -y;   // Sets x equal to negative of y
  - Other examples of unary operators:
    x = ++y;

    x = --y;
  - Unary operators can also be overloaded

Overload "-" for Money

- **Overloaded "-" function declaration**
  - Placed outside class definition:
    const Money operator –(const Money& amount);
  - Notice only one argument since only one operand

- **"-" operator can be overloaded twice**
  - For two operands/arguments (as a binary operator)
  - For one operand/argument (as a unary operator)
  - **Definitions must exist for both**

Overloaded "-" Definition

- **Overloaded "-" function definition:**

  -
    ```
    const Money operator –(const Money& amount)
    {
    return Money(-amount.getDollars(),
                                 -amount.getCents());
    }
    ```
  - Applies "-" unary operator to the built-in type
  - **Returns anonymous object again**

Overloaded "-" Usage

- **Consider:**

  -
    ```
    Money   amount1(10),
            amount2(6),
            amount3;
    amount3 = amount1 – amount2; // calls binary "-" overload
    ```

  -
    ```
    amount3.output(); //Displays $4.00
    amount3 = -amount1;       // Calls unary "-" overload
    ```

  -
    ```
    amount3.output()  //Displays -$10.00
    ```

Overloading as Member Functions

- **In previous examples, the operators were standalone functions, defined outside a class**
  - We can also overload them as member operators
  - And then consider them as member functions like others

- **When operator is member function:**
  - Only one parameter needs to be passed, not two
  - **Calling the object itself serves as the first parameter**
  - Example: *Money cost(1, 50), tax(0, 15), total; total = cost + tax;*
  - If "+" overloaded as member operator: object cost is the calling object and Object tax is a single argument
  - **Think of as: total = cost.+(tax);**

- **Declaration of "+" in class definition:**
  - const Money operator +(const Money& amount);

Other Overloads

- **&&, ||, and comma operator**
  - Predefined versions work for bool types
  - Recall that these use short-circuit evaluation
  - When overloaded no longer uses short-circuit by default
  - Uses complete evaluation instead, which may be contrary to expectations
  - **Generally one should not need to overload these operators**

Friend Functions

- **Special category of non-member functions**
  - Recall that operator overloads is typically declared as a non-member function
  - Hence, they access data through accessor and mutator methods, thereby suffering the overhead of calls
  - Friends can directly access private class data, so there is no calling overhead involved
  - **Simply put, declaring non-member operators as friends can improve their performance**

- **Use keyword friend in front of the function declaration**
  - Specified in the class declaration
  - But not treated as a member function

Friend Function Purity

- **Friend functions are not compliant with OOP?**
  - The OOP principles dictate that all operators and functions be member functions
  - Therefore, friend functions violate the purity of the basic OOP principles for the purposes of run-time efficiency

- **Why consider them then?**
  - Advantageous for operator overloading and their efficiency
  - Still follow encapsulation since a friend function is in the class declaration

Friend Classes

- **Entire classes can be friends**
  - Similar to function being friend to class
  - Example:
    class F is friend of class C
  - All class F member functions are friends of C
  - **However, this is not reciprocated**
    (i.e., friendship can be granted but not taken) ☹

- **Syntax:**
  - *friend class F*
  - Goes inside the class declaration of the authorizing class

References

- **Reference defined:**
  - Name of a storage location
  - Similar to a pointer, which will be discussed later

- **Example of stand alone reference:**
  - *int robert;*
    int& bob = robert;
  - bob is now reference to storage location for robert
  - Changes made to bob will affect robert

- **Useful in several cases:**
  - Call-by-reference, as discussed so far
  - Returning a reference, where an alias to a variable is returned instead of a new variable

Returning a Reference

- **Syntax:**
  - *double& sampleFunction(double& variable);*
  - double& and double in declaration are different

- **Returned item must have a reference**
  - Like a variable of that type
  - Cannot be expression like "x+5” since this has no place in memory to reference

- **Example function definition:**

  -
    ```
    double& sampleFunction(double& variable) {
            return variable;
    }
    ```
  - Mainly used to implement overloaded operators

Overloading >> and <<

- **Enables input and output of our objects**
  - Similar to other operator overloads
  - Improves readability, similar to the purpose of other operator overloads

- **Enables:**
  - *cout << myObject;*
    cin >> myObject;

- **Instead of the special output functions such as:**
  - *myObject.output();*

Overloading <<

- **Insertion operator, <<**
  - Used with cout as a binary operator

- **Example:**
  - cout << "Hello";
  - **The first operand is predefined object cout**
  - The second operand is literal string "Hello“

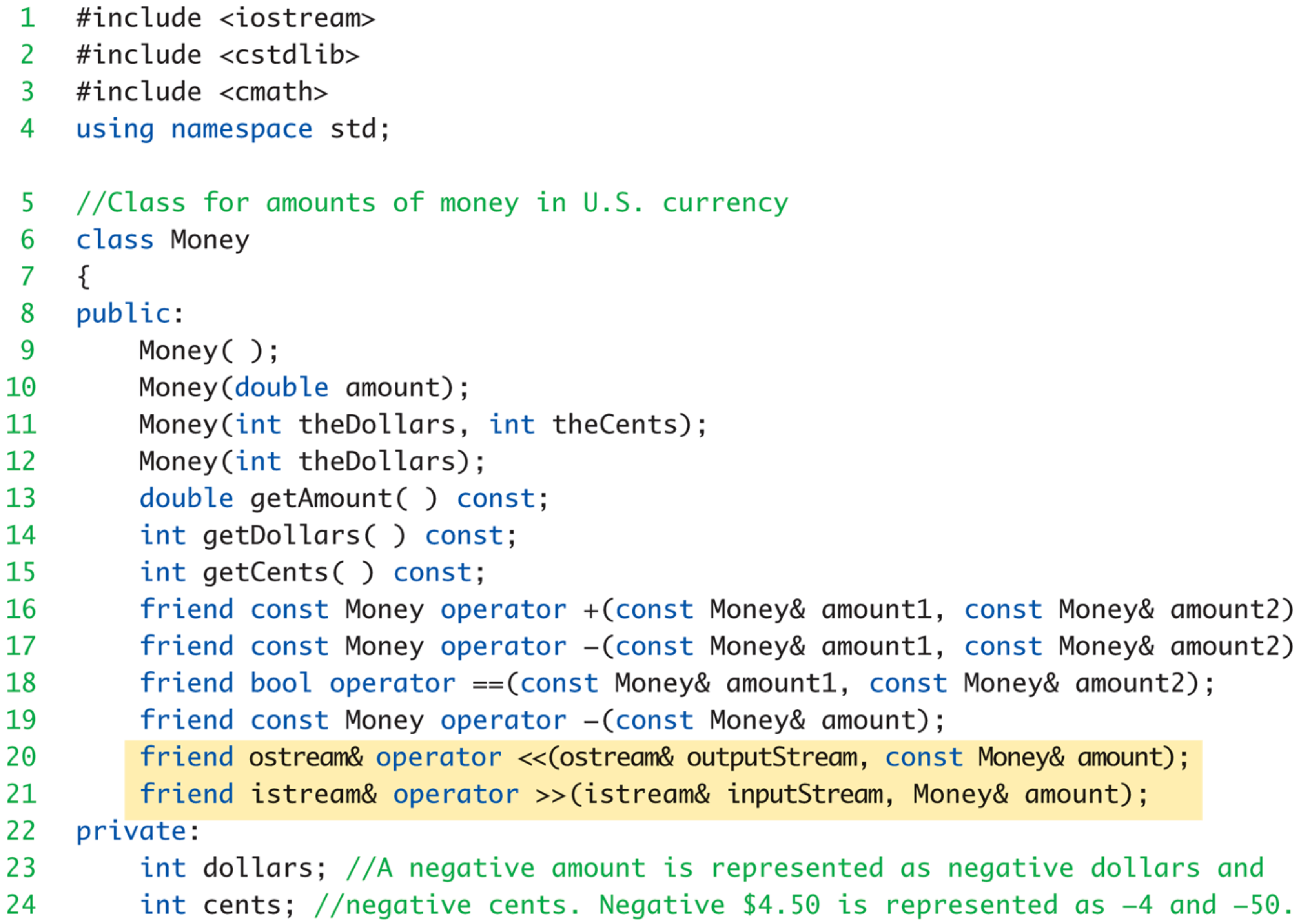
- **Recall Money class**
  - Nicer if we can use << operator:
  - *Money amount(100);*
    cout << "I have " << amount << endl;
  - instead of:
    cout << "I have “; amount.output()

Overloaded << Return Value

- **Example:**
  - *Money amount(100);*
    cout << amount;
  - << should return some value

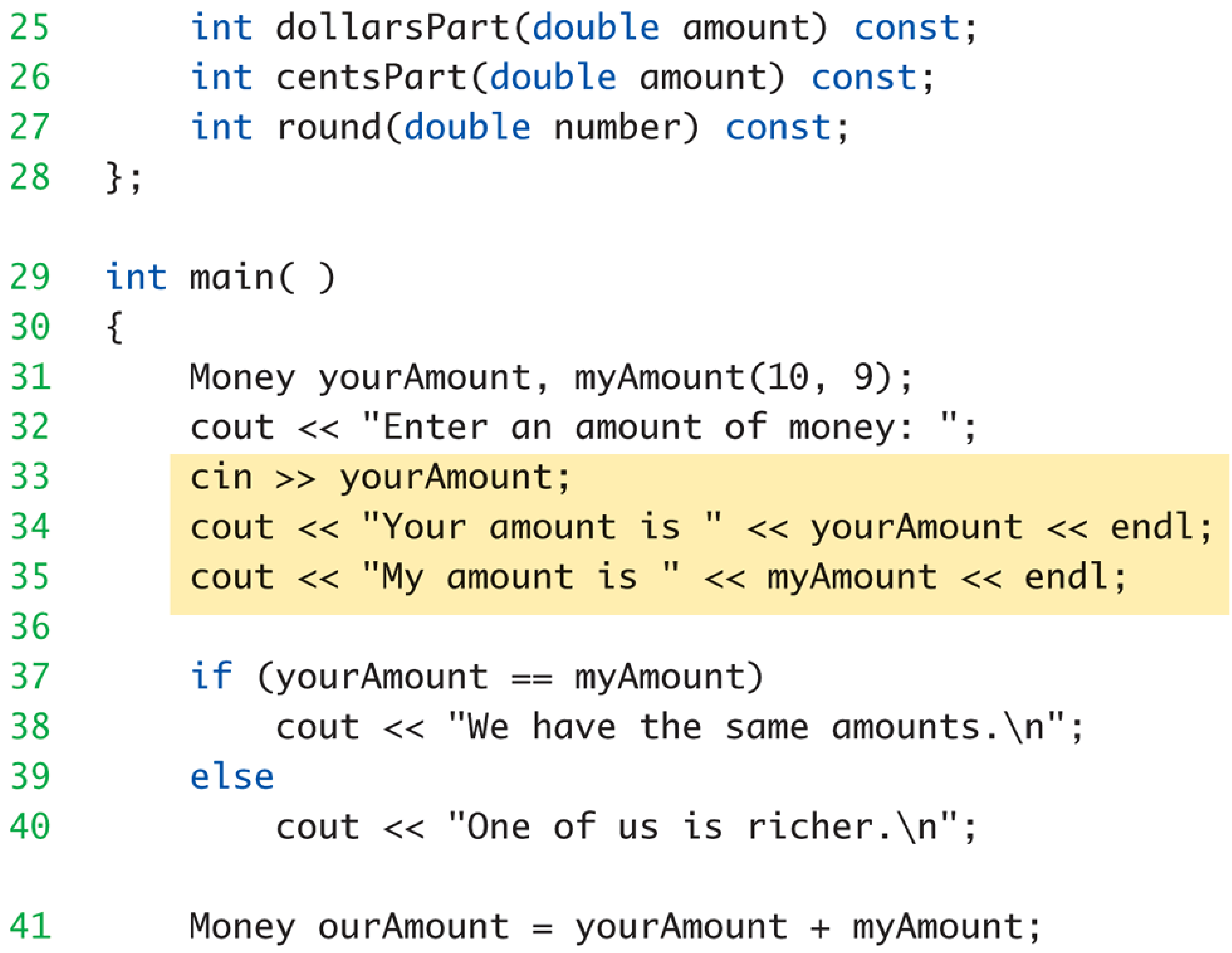
- **How do we allow operator cascading such as:**
  - *cout << "I have " << amount;*
    (cout << "I have ") << amount;
  - Return an instance of a cout object
  - That is, returns its first argument type: ostream

Overloaded << Example /1



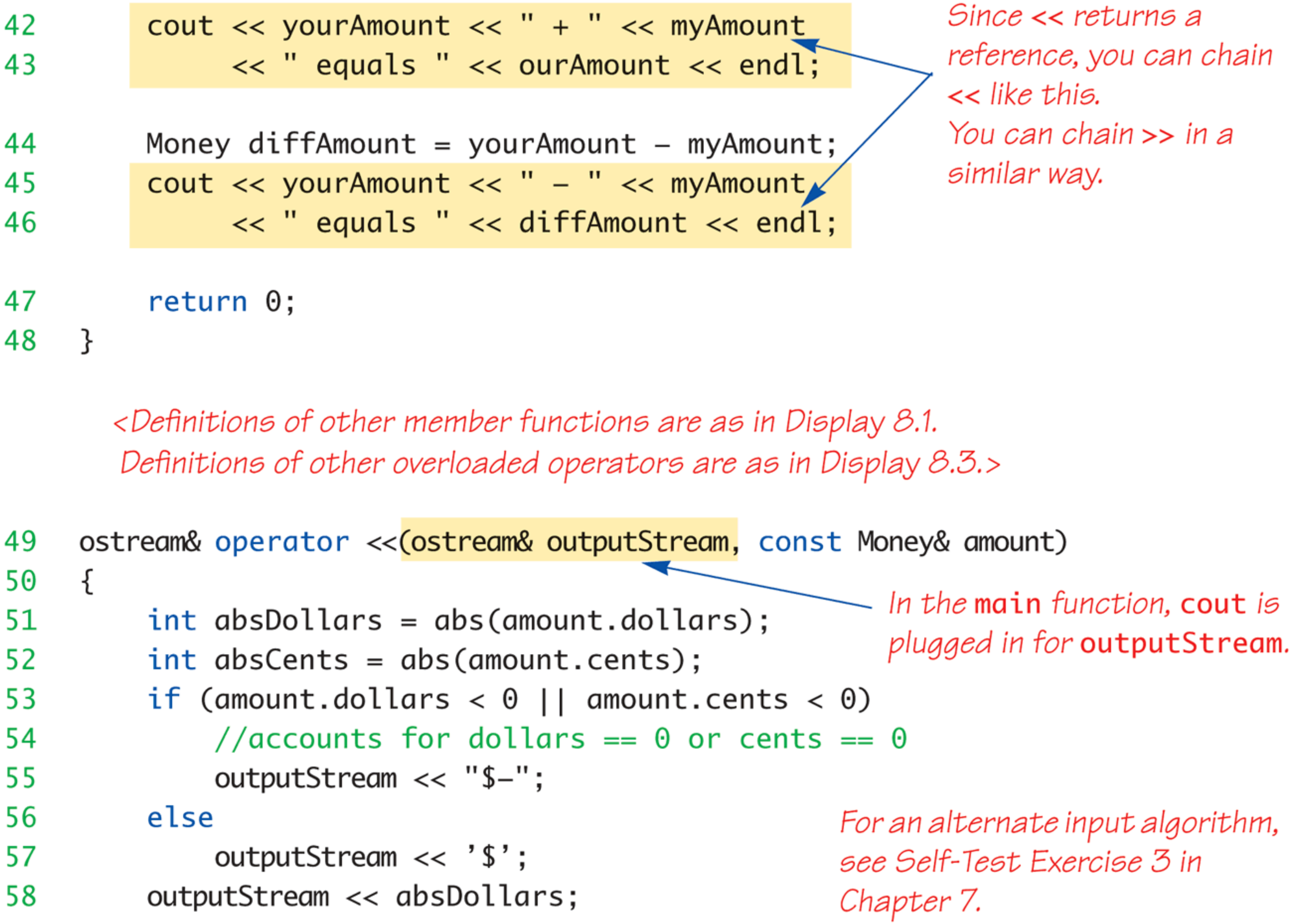
```
#include <iostream>
#include <cstdlib>
#include <cmath>
using namespace std;

//Class for amounts of money in U.S. currency
class Money
{
public:
    Money( );
    Money(double amount);
    Money(int theDollars, int theCents);
    Money(int theDollars);
    double getAmount( ) const;
    int getDollars( ) const;
    int getCents( ) const;
    friend const Money operator +(const Money& amount1, const Money& amount2)
    friend const Money operator -(const Money& amount1, const Money& amount2)
    friend bool operator ==(const Money& amount1, const Money& amount2);
    friend const Money operator -(const Money& amount);
    friend ostream& operator <<(ostream& outputStream, const Money& amount);
    friend istream& operator >>(istream& inputStream, Money& amount);
private:
    int dollars; //A negative amount is represented as negative dollars and
    int cents; //negative cents. Negative $4.50 is represented as -4 and -50.
```

Overloaded << Example /2



```
    int dollarsPart(double amount) const;
    int centsPart(double amount) const;
    int round(double number) const;
};

int main( )
{
    Money yourAmount, myAmount(10, 9);
    cout << "Enter an amount of money: ";
    cin >> yourAmount;
    cout << "Your amount is " << yourAmount << endl;
    cout << "My amount is " << myAmount << endl;

    if (yourAmount == myAmount)
        cout << "We have the same amounts.\n";
    else
        cout << "One of us is richer.\n";

    Money ourAmount = yourAmount + myAmount;
```

Overloaded << Example /3



```
    cout << yourAmount << " + " << myAmount
         << " equals " << ourAmount << endl;

    Money diffAmount = yourAmount - myAmount;
    cout << yourAmount << " - " << myAmount
         << " equals " << diffAmount << endl;

    return 0;
}
```

Since << returns a reference, you can chain << like this. You can chain >> in a similar way.

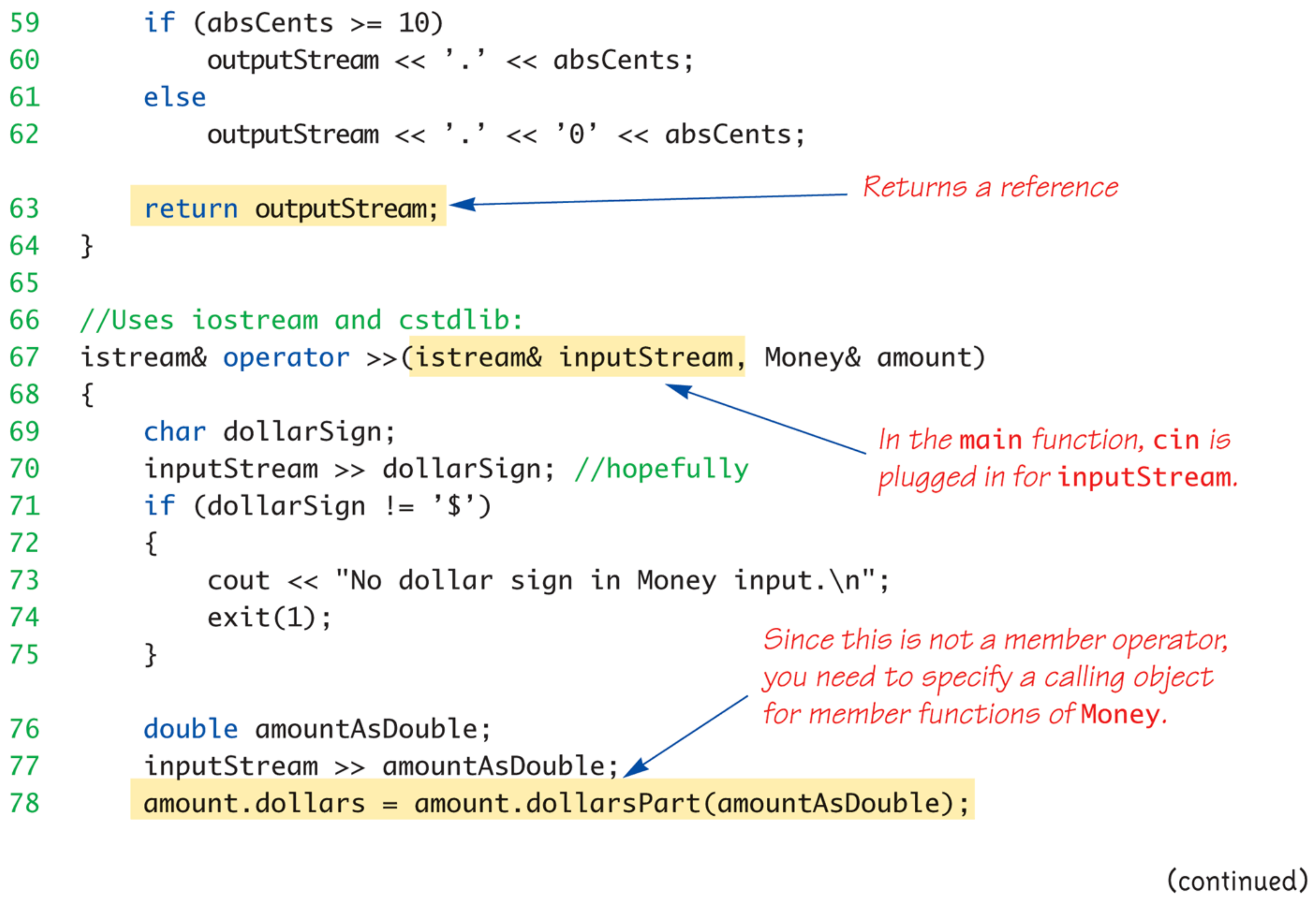
<Definitions of other member functions are as in Display 8.1.
Definitions of other overloaded operators are as in Display 8.3.>

```
ostream& operator <<(ostream& outputStream, const Money& amount)
{
    int absDollars = abs(amount.dollars);
    int absCents = abs(amount.cents);
    if (amount.dollars < 0 || amount.cents < 0)
        //accounts for dollars == 0 or cents == 0
        outputStream << "$-";
    else
        outputStream << '$';
    outputStream << absDollars;
```

In the **main** function, **cout** is plugged in for **outputStream**.

For an alternate input algorithm, see Self-Test Exercise 3 in Chapter 7.

Overloaded << Example /4



```
    if (absCents >= 10)
        outputStream << '.' << absCents;
    else
        outputStream << '.' << '0' << absCents;
    return outputStream;
}

//Uses iostream and cstdlib:
istream& operator >>(istream& inputStream, Money& amount)
{
    char dollarSign;
    inputStream >> dollarSign; //hopefully
    if (dollarSign != '$')
    {
        cout << "No dollar sign in Money input.\n";
        exit(1);
    }
    double amountAsDouble;
    inputStream >> amountAsDouble;
    amount.dollars = amount.dollarsPart(amountAsDouble);
```

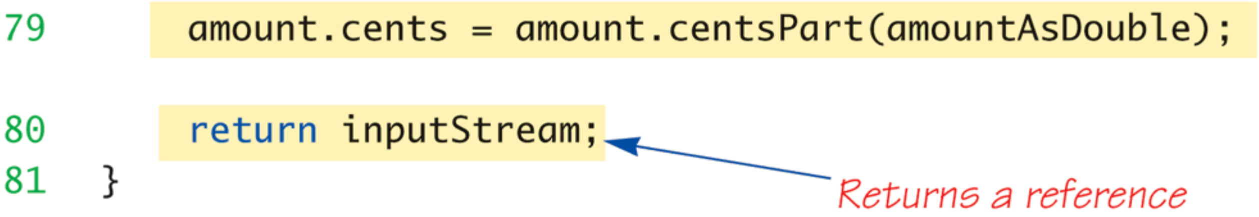
Returns a reference

In the **main** *function,* **cin** *is plugged in for* **inputStream***.*

Since this is not a member operator, you need to specify a calling object for member functions of **Money***.*

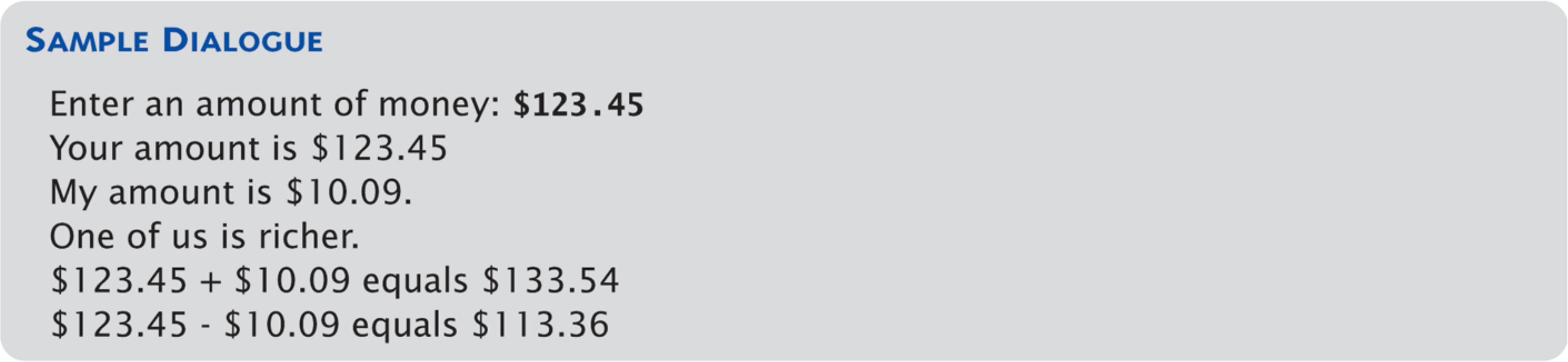
(continued)

Overloaded << Example /5



```
        amount.cents = amount.centsPart(amountAsDouble);

        return inputStream;
    }
```

Returns a reference



Sample Dialogue

```
Enter an amount of money: $123.45
Your amount is $123.45
My amount is $10.09.
One of us is richer.
$123.45 + $10.09 equals $133.54
$123.45 - $10.09 equals $113.36
```

Assignment Operator, =

- **Must be overloaded as a member operator**
  - Automatically overloaded by the compiler
  - Works as a default assignment operator
  - That is, as a member-wise copy, where member variables from one object are copied into the corresponding member variables from other
  - With pointers, need to write your own version

- **Overload Array Operator, [ ]**
  - Can overload [ ] for the specific class type
  - To be used to iterate objects of the class type
  - The operator must return a reference
  - And the operator [ ] must be a member function