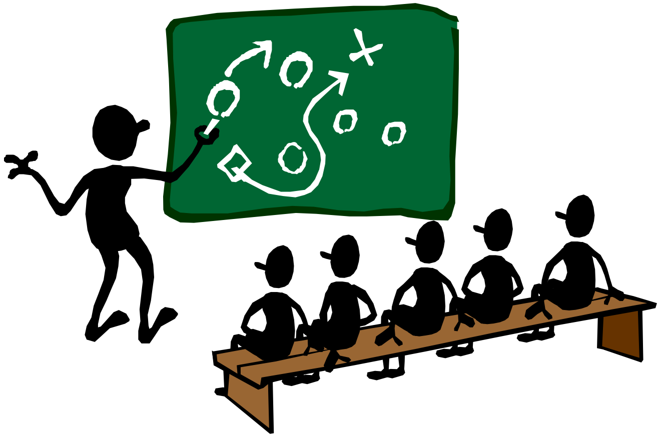
# C++ Programming Language
# Chapter 7  Constructors & Other Tools



*Juinn-Dar Huang*

*Associate Professor*

*jdhuang@mail.nctu.edu.tw*

*March 2011*

# Learning Objectives

- Constructors (ctors)
  - definitions and calls
  - overloaded ctors
  - default ctor and copy ctor
- Class objects as data members
- More class constructs
  - const member functions
  - inline member functions
  - static data members
  - static member functions

# Constructors (ctors)

- ctor is dedicated to
  - initialization of some or all data members
  - other necessary actions during initialization
- ctor is a special kind of member function
  - **automatically** called when an object is **born**
    - it's automatic so it's impossible to forget initializing an object
- ctor is declared just like any other member functions except
  - ctor name MUST be SAME as class name
  - ctor has NO return type; not even void
- Yes, there are destructors (dtors), too
  - automatically called when objects are dead
  - will be discussed in Chapter 10

# Constructor Declaration

- Class definition with ctor declaration

optional

same

no return type

```
class DayOfYear {
public:
    DayOfYear(int monthValue, int dayValue); // ctor
    void input();
    void output();
    // …
private:
    int month;
    int day;
}
```

# Calling Constructors (1/2)

```
void f() {
        DayOfYear date1(7, 4), date2(3, 6);
        // …
}
```

implicit ctor call
for date1 with (7, 4)

implicit ctor call
for date2 with (3, 6)

- As soon as data1/date2 is born (created)
  - – ctor is automatically called for each
  - – values in parentheses passed as arguments to ctor
  - – data members (month & day) are then initialized by ctor (you will see later)

# Calling Constructors (2/2)

```
void f() {
    DayOfYear date1;             // Error, date1 must call ctor
    date1.DayOfYear(7, 4);       // Error, cannot call ctor directly
    DayOfYear date2(3, 6);       // ok
    // …
 }
```

- Error
  - – if there are any existing ctors, it is **NOT** allowed to define (create) an uninitialized object by not calling ctors
- Error
  - – object is NOT allowed to call ctors directly

# Reminders for Constructors

- Same name as class itself
- ctor has no return type
  - not even void!
- ctor is automatically called when object is created
- Object must be initialized by ctor if there are any
- Object cannot call ctors directly
- ctors are public in most cases
  - mostly, objects are born outside class
  - but they can be private (beyond the scope of this course)

# Constructor Definition

- ctor definition is like all other member functions
  - – except it has no return type

class name ↓ function name ↓

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

- Note that same name around ::
  - – clearly identifies a ctor
- Note that no return type
  - – just as in class definition

# Complete Constructor Definition (1/2)

```
DayOfYear::DayOfYear(int monthValue, int dayValue)
      : month(monthValue), day(dayValue)
{     // can be empty if nothing to do here    }
```

- is called member initializer list
  - MUST start with a colon
  - data members are separated by commas
  - actually, data members are **ALWAYS** initialized here
- **Prefer this style for ctor definition**

P17

# Complete Constructor Definition (2/2)

- Suggestion: Use member initializer list for data member initialization whenever possible

- Function body of ctor can be empty
  - if there is nothing to do indeed

- Function body of ctor can also be non-empty
  - e.g., check validity of input parameters

# Overloaded Constructors

- ctor can be overload just like other functions
  - that is, a class can have multiple ctors

```
class DayOfYear {
public:
    DayOfYear(int monthValue, int dayValue); // ctor #1
    DayOfYear(int monthValue) // ctor #2, dayValue = 1 by default
   DayOfYear( )   // ctor #3, yearValue = dayValue = 1 by default
    // …
}
void f() { DateOfYear d1(7, 4), d2(9), d3; … }
```

- Recall: a function signature consists of:
  - function name
  - parameter list
- Provide constructors for all possible initialization ways

# Default Constructors (1/2)

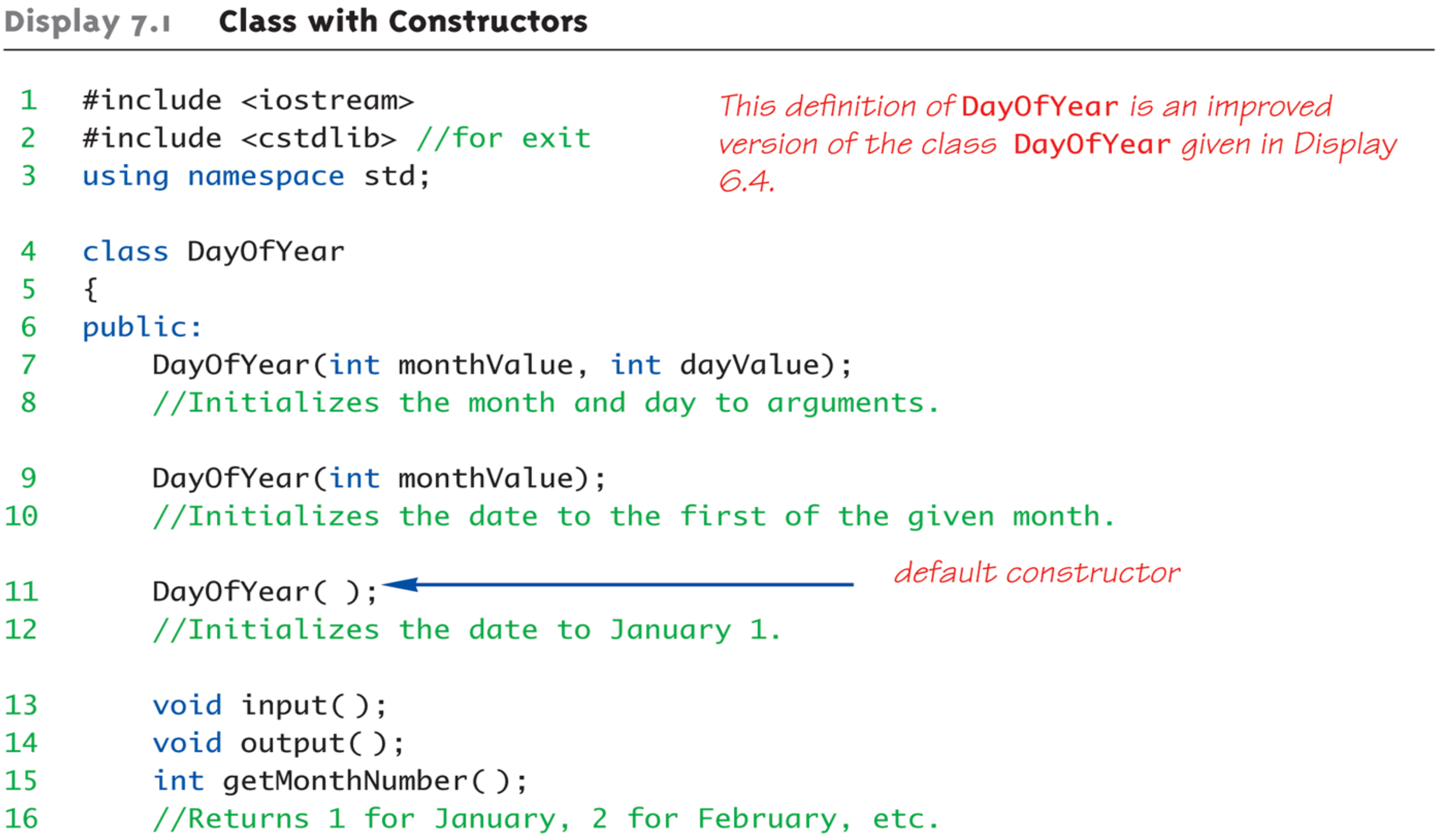
- Default ctor
  - is a ctor that can be called **without supplying arguments**
  - e.g., ctor #3 in previous slide
- Object definition with no arguments for ctor:
  - DayOfYear date1;    // no arguments for ctor ➔ call **default ctor!**
  - DayOfYear date2(); // NO! compilation error
    - why?
    - compiler sees a function declaration here!
    - yes, it is confusing
- Standard functions with no arguments:
  - called with syntax: callMyFunction();
    - including empty parentheses

# Default Constructors (2/2)

For a class

- If there are one or more user-defined ctors
  - one of those ctors MUST be called when an object is created
  - it is programmer's responsibility to provide appropriate arguments; or, it will be a compilation error
  - if there is a user-defined default ctor, an object can be created without providing any arguments
- If there are **NO** user-defined ctors at all **(advanced)**
  - compiler will generate a default ctor if needed
  - the generated default ctor
    - implicitly calls the defaults ctors for all data members of class types (we will see this soon)
    - implicitly calls the default ctors of bases (we will see this in Chap 14)
- **Pitfall**: if you provide any ctors but no default ctor
  ➔ compiler won't generate a default ctor for you
  ➔ you cannot create an object w/o supplying arguments

# Class with Constructors Example (1/3)



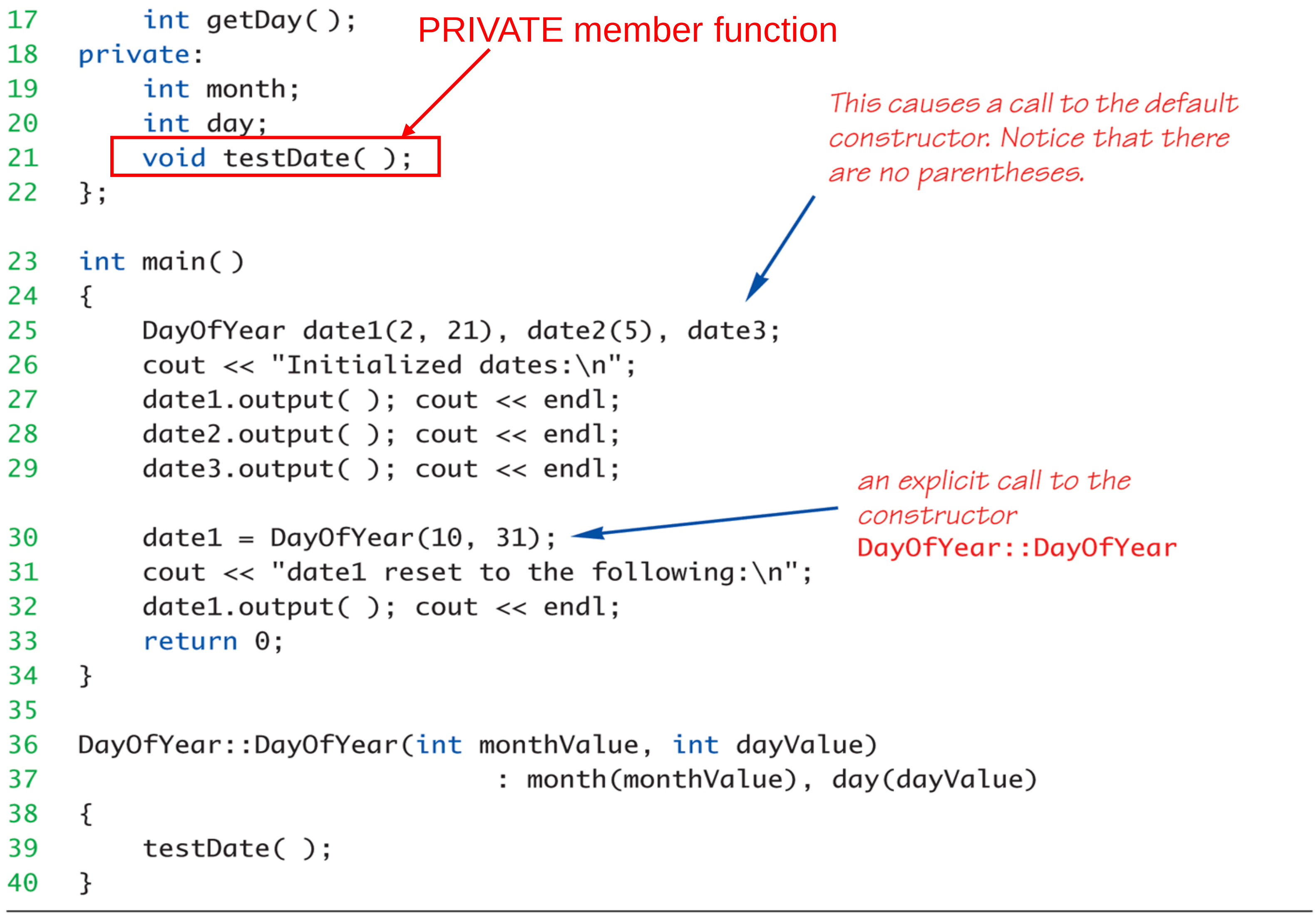
**Display 7.1 Class with Constructors**

```
#include <iostream>
#include <cstdlib> //for exit
using namespace std;

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.
```

*This definition of* **DayOfYear** *is an improved version of the class* **DayOfYear** *given in Display 6.4.*

*default constructor*

# Class with Constructors Example (2/3)



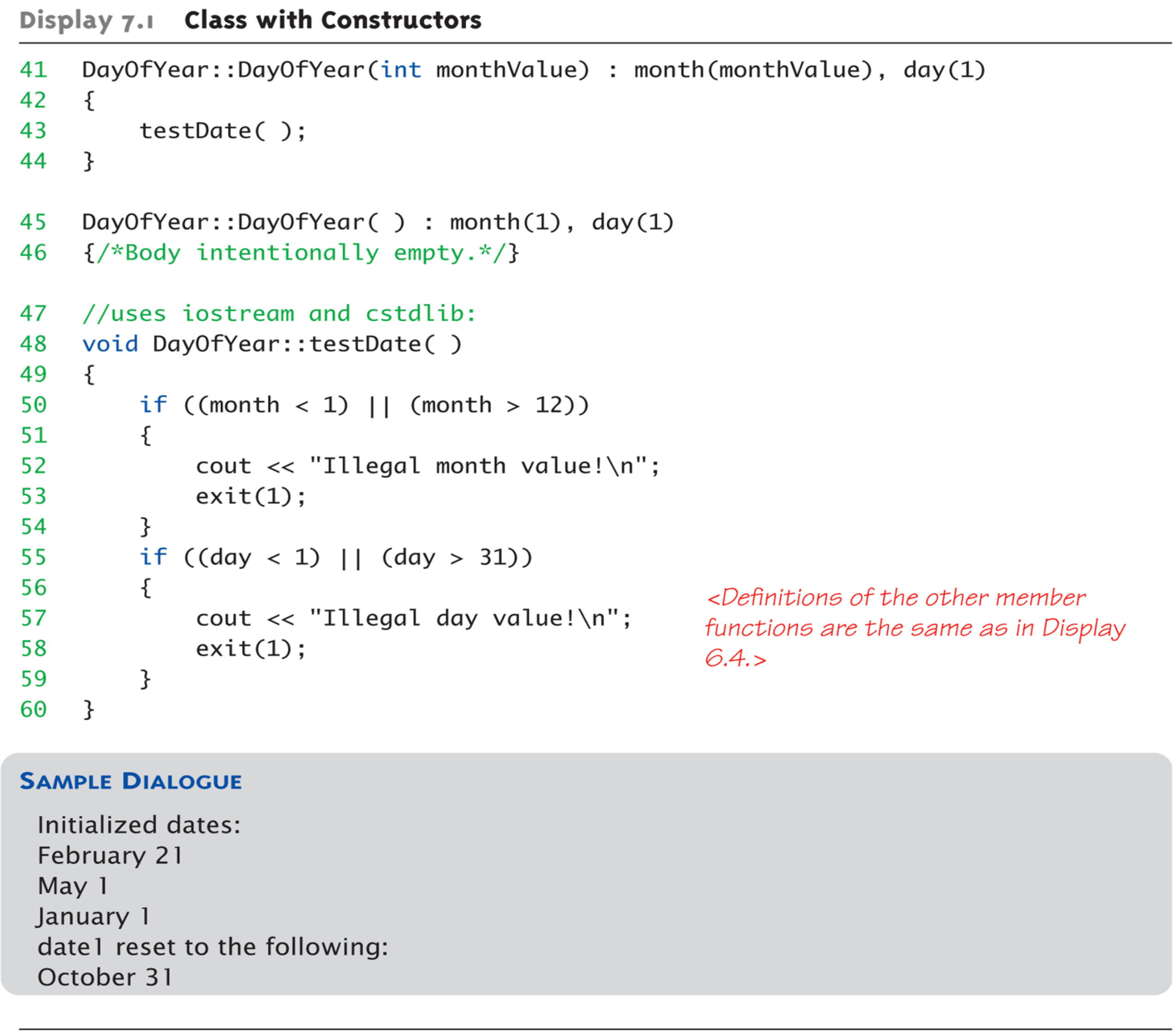
PRIVATE member function

```
    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.*

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

# Class with Constructors Example (3/3)



**Display 7.1 Class with Constructors**

```
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
```

# Copy Constructor (1/2)

- The form of copy ctor of a class X ➔ X::X(const X&)
  - use an existing object of class X for initialization

```
DayOfYear::DayOfYear(const DayOfYear& src)
: month(src.month), day(src.day) { }
void f() {
    DayOfYear date1(5, 5);
    DayOfYear date2(date1);       // use copy ctor; prefer this one!
    DayOfYear date3 = date2;      // STILL use copy ctor!
    date3 = date1;                // use assignment operator
}
```

- If you do not define a copy ctor for a class
  - compiler will make a default one for you!
  - the default copy ctor performs **member-wise copy**

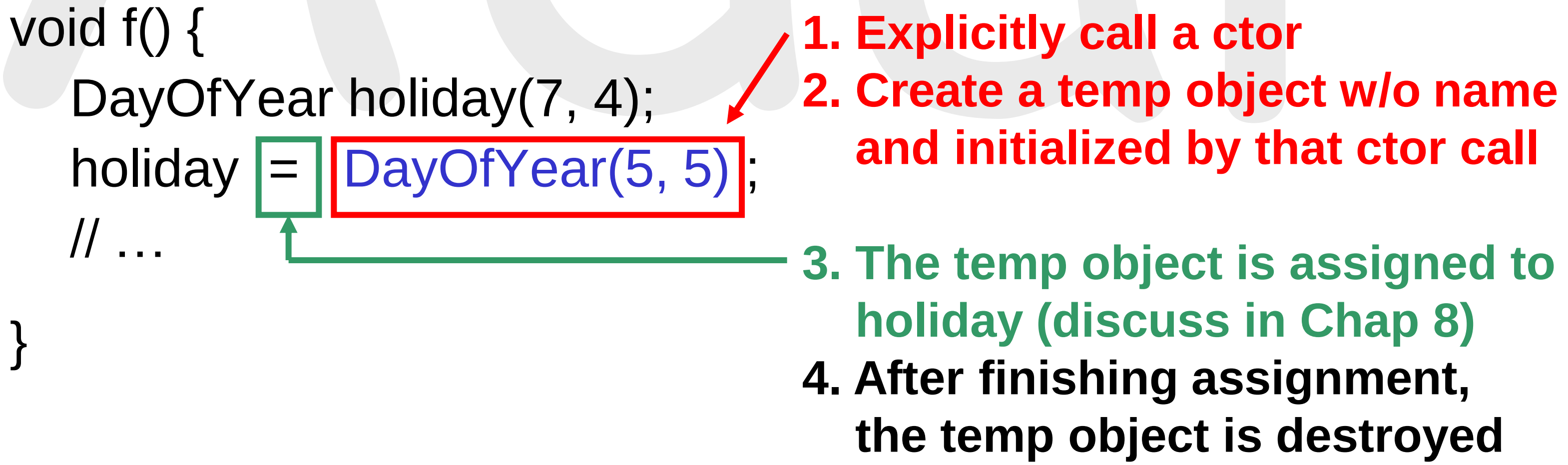
# Copy Constructor (2/2)

- Applications of copy ctor
  - more than you could expect!
  - variable initialization (1); argument passing (2); value return (3)

```
// src: call-by-value parameter
complex g(complex src) {  // src ← a (Case 2)
    complex result(src);   // result ← src (Case 1)
    result += src;
    return result;   // return value (a temporary object)  t ← result (Case 3)
}
void f() {
    complex a(1,1), b;
    b = g(a);          // assignment operator: b = t
}
```

# Explicit Constructor Calls (Advanced)

- ctors can be explicitly called
- A temporary object will be created by explicitly calling a ctor
  - that object has no name
  - it is destroyed (i.e., dead) at the end of expression in which it was created



```
void f() {
    DayOfYear holiday(7, 4);
    holiday = DayOfYear(5, 5);
    // …

}
```

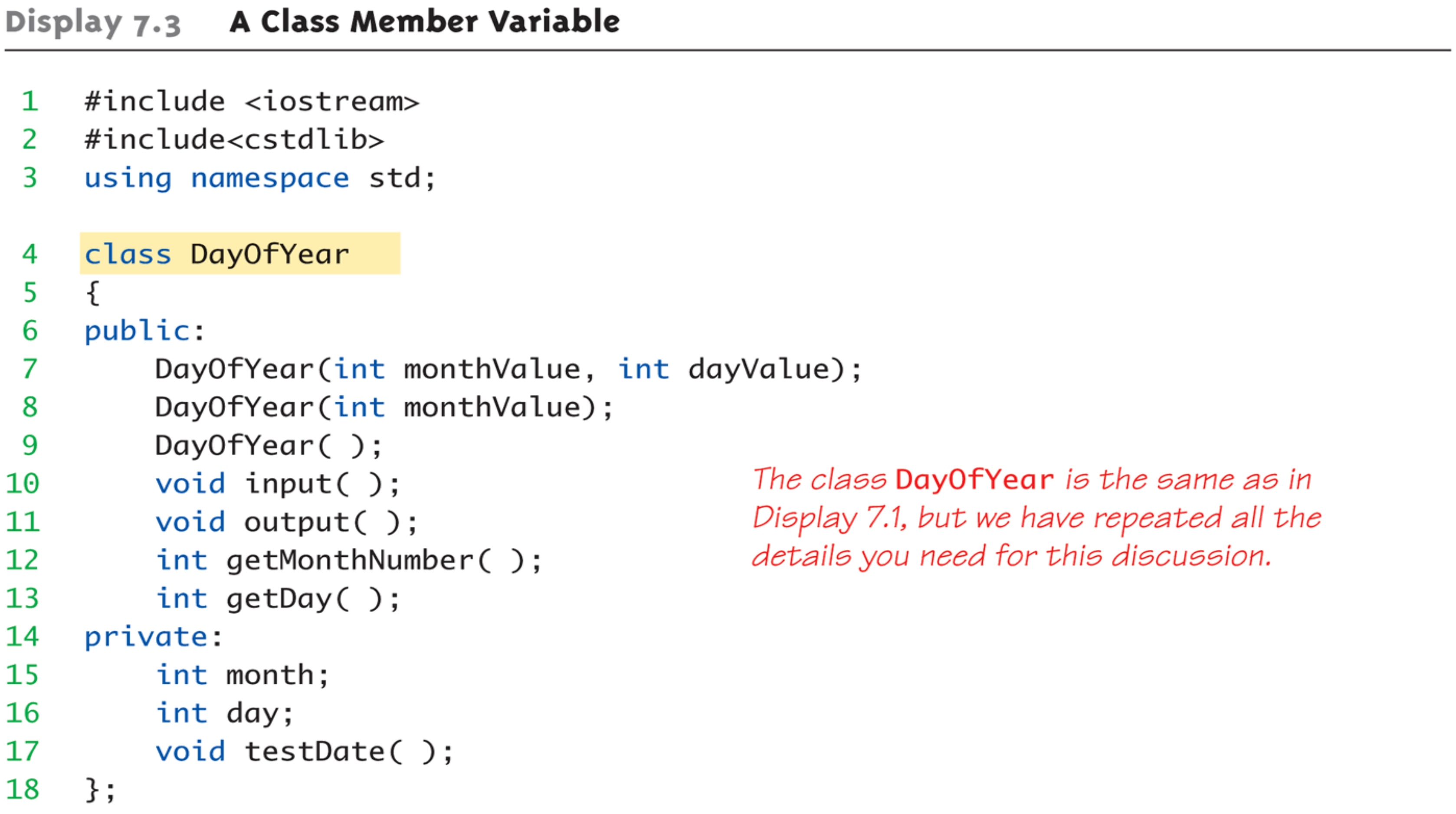
1. Explicitly call a ctor
2. Create a temp object w/o name and initialized by that ctor call
3. The temp object is assigned to holiday (discuss in Chap 8)
4. After finishing assignment, the temp object is destroyed

# Class Objects as Data Members

- Class data member can be any type
  - including class types as well
- For class data members
  - call their own ctors ONLY in member initializer list
    - there is no other place better than here; think about it
    - hence, they can be properly initialized before being used
- A class data member without default ctor
  - must be explicitly initialized in member initializer list
- A class data member with default ctor
  - ok to be absent in member initializer list (default ctor is called)

P8

# Class Member Variables (1/5)

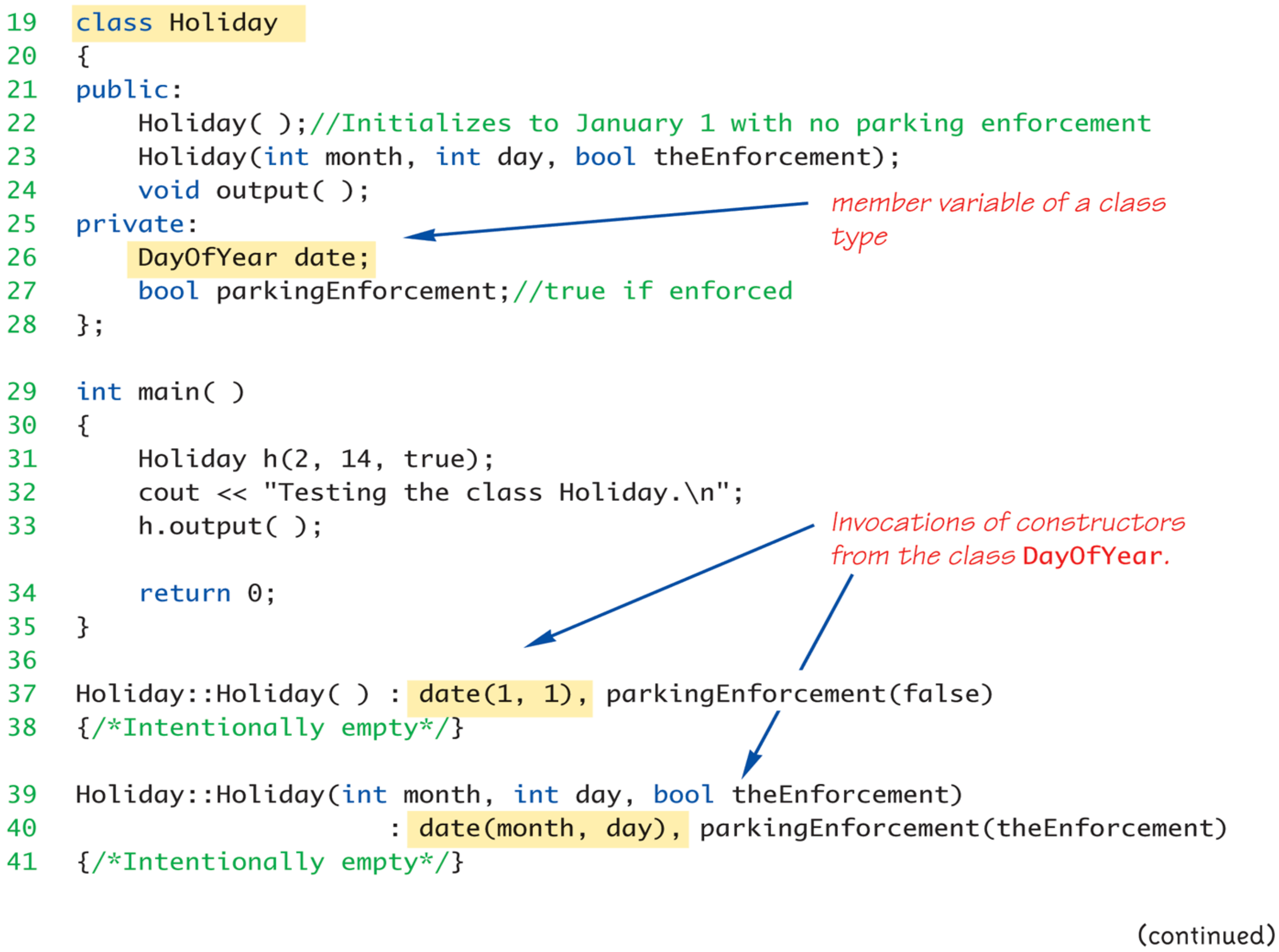


**Display 7.3 A Class Member Variable**

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

class DayOfYear
{
public:
    DayOfYear(int monthValue, int dayValue);
    DayOfYear(int monthValue);
    DayOfYear( );
    void input( );
    void output( );
    int getMonthNumber( );
    int getDay( );
private:
    int month;
    int day;
    void testDate( );
};
```

*The class* **DayOfYear** *is the same as in Display 7.1, but we have repeated all the details you need for this discussion.*

# Class Member Variables (2/5)



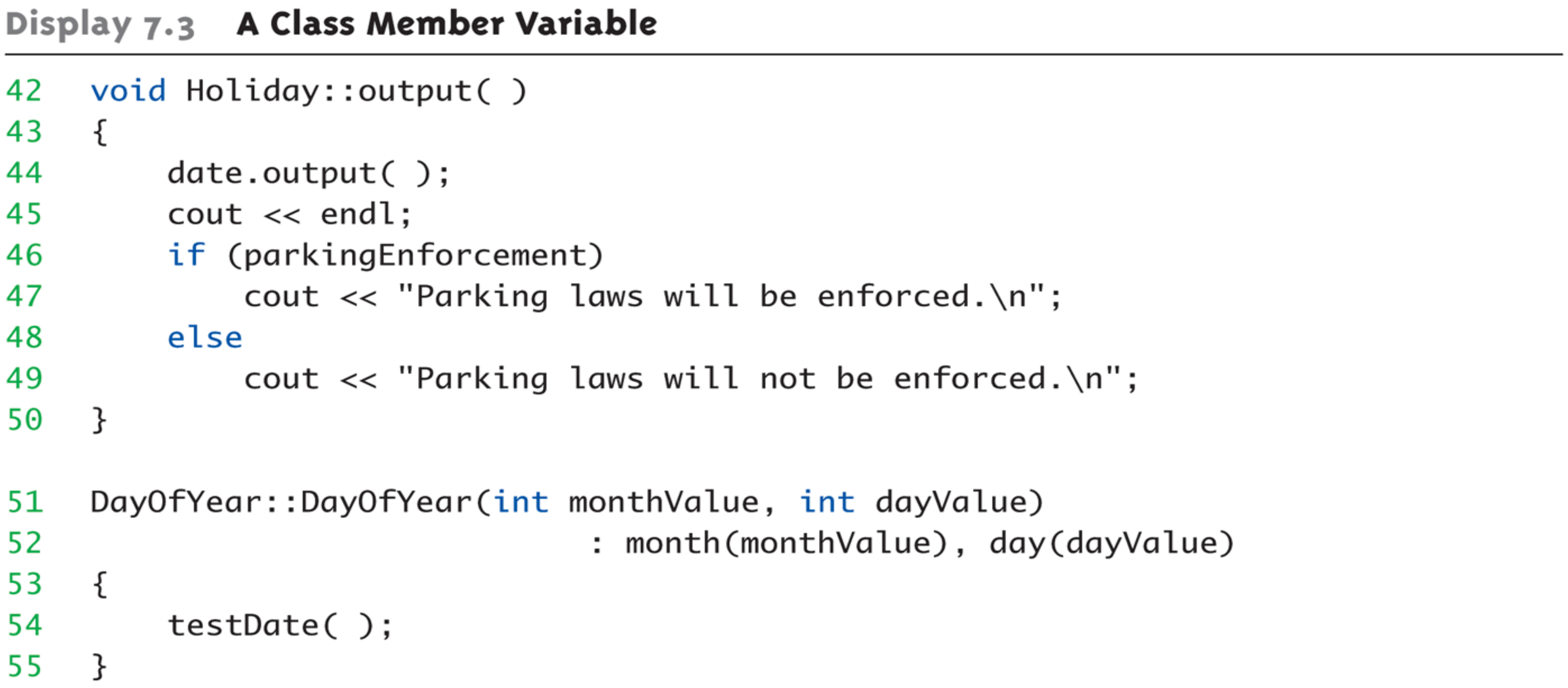
```
class Holiday
{
public:
    Holiday( );//Initializes to January 1 with no parking enforcement
    Holiday(int month, int day, bool theEnforcement);
    void output( );
private:
    DayOfYear date;
    bool parkingEnforcement;//true if enforced
};

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**.

(continued)

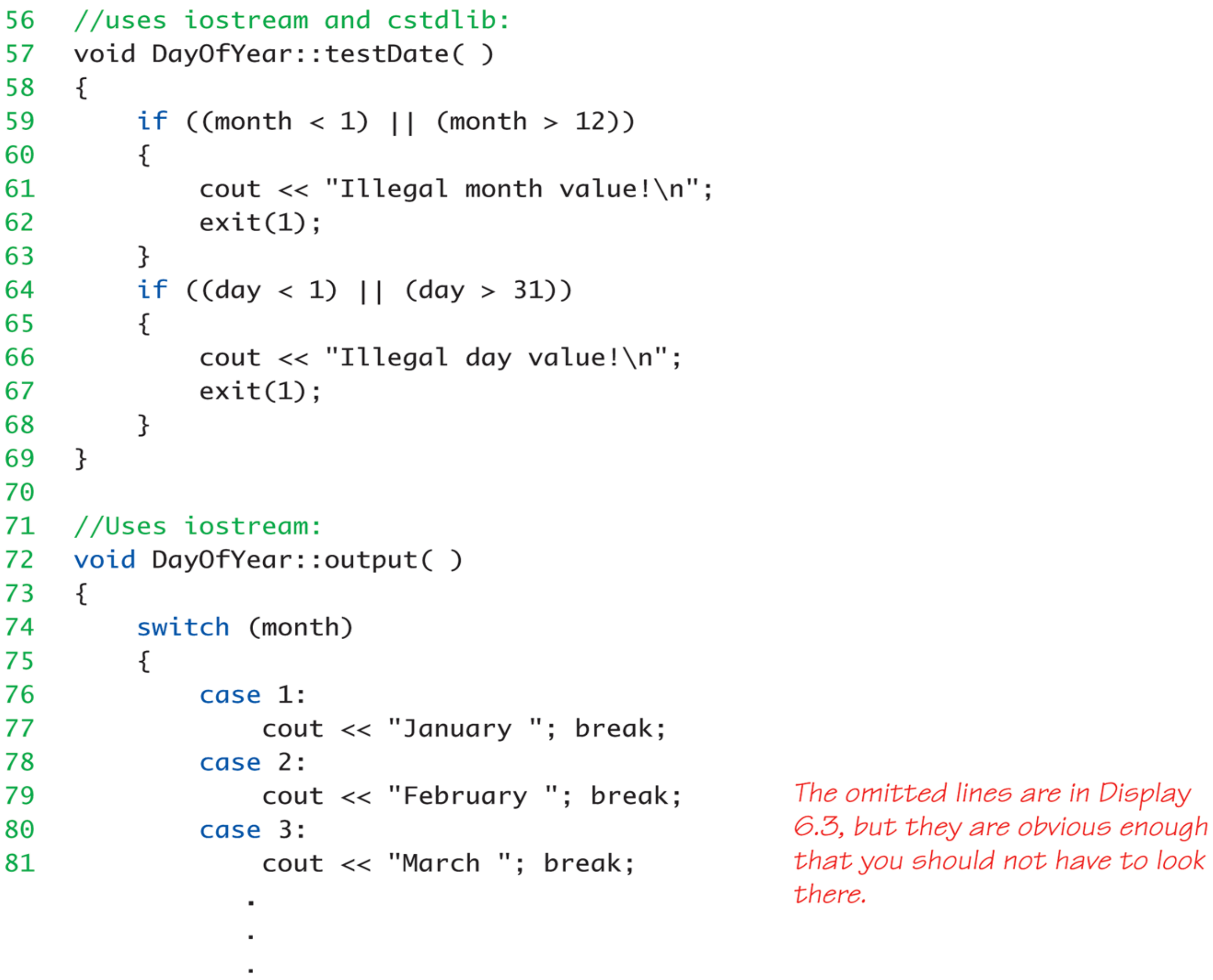
# Class Member Variables (3/5)



**Display 7.3 A Class Member Variable**

```
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";
}

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

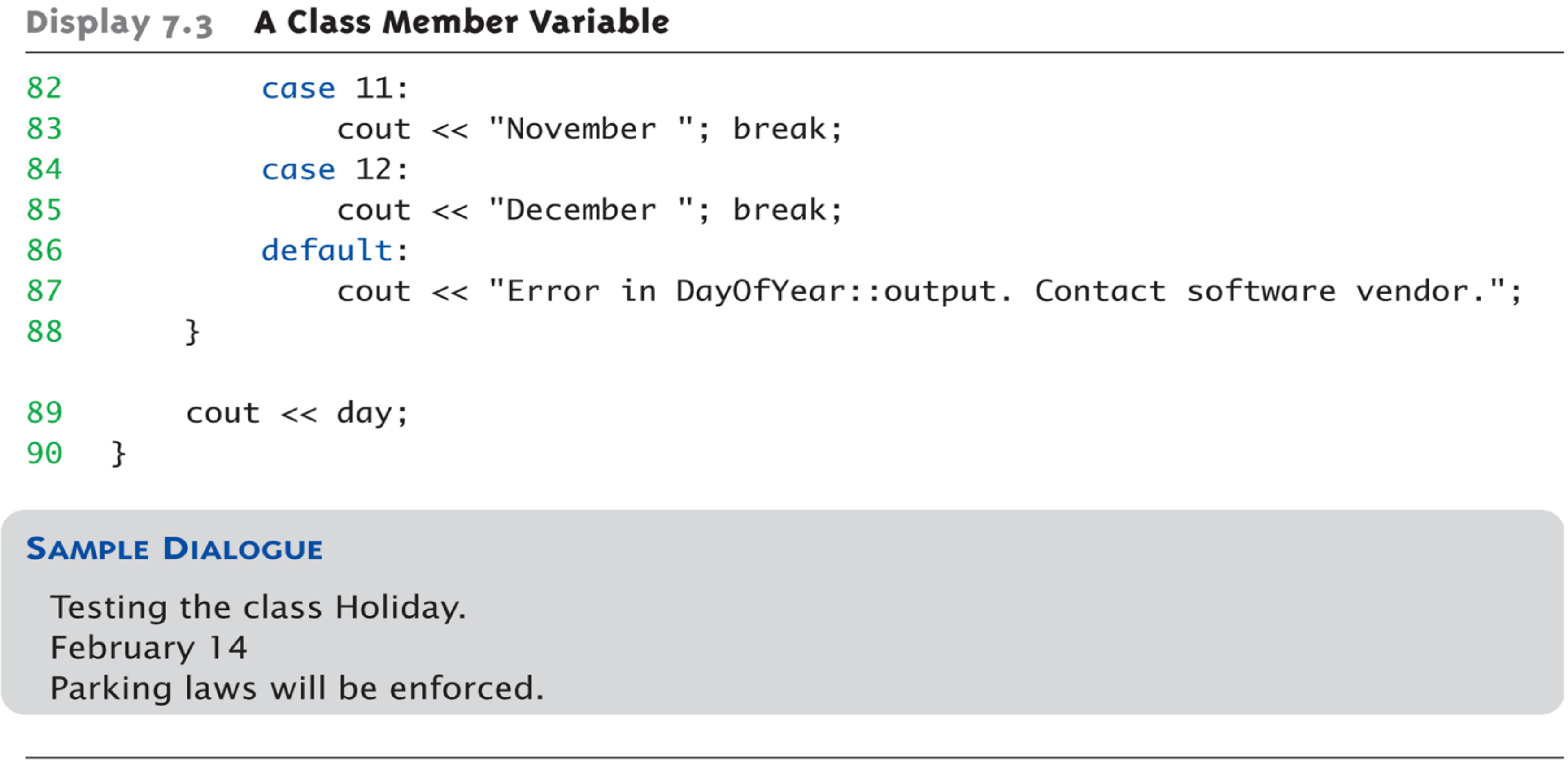
# Class Member Variables (4/5)



```
//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);
    }
}

//Uses iostream:
void DayOfYear::output( )
{
    switch (month)
    {
        case 1:
            cout << "January "; break;
        case 2:
            cout << "February "; break;
        case 3:
            cout << "March "; break;
               .
               .
               .
```

*The omitted lines are in Display 6.3, but they are obvious enough that you should not have to look there.*

# Class Member Variables (5/5)



Display 7.3 A Class Member Variable

```
            case 11:
                cout << "November "; break;
            case 12:
                cout << "December "; break;
            default:
                cout << "Error in DayOfYear::output. Contact software vendor.";
        }

        cout << day;
    }
```

**Sample Dialogue**

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

# Parameter Passing in Functions (1/2)

- Efficiency of parameter passing
  - call-by-value
    - requires copy being made ➔ runtime and memory overhead
  - call-by-reference
    - alias for actual argument
    - more efficient way
  - negligible difference for basic data types (int, double, …)
  - for BIG class objects (e.g., 1MB) ➔ clear advantage

As we discussed previously, the conventions are:

- Read-only ➔ call-by-const-reference is desirable
- Modification possible ➔ call-by-pointer-value is desirable

# Parameter Passing in Functions (2/2)

- If a function does not intend to make modifications through its pointer or reference parameters
  - protect them with const
  - protect ALL such parameters
- This includes parameters of class member function

# Constant Member Functions (1/2)

- If a member function does not make any modifications on data members
  - **ALWAYS** make it a constant member function
- Revisit example @ P18~P22

```
class DayOfYear {
public:
    DayOfYear(int, int);
    DayOfYear(int);
    DayOfYear( );
    void output( ) const;
    int getMonthNumber( ) const;
    int getDay( ) const;
```

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

class Holiday {
public:
    Holiday( );
    Holiday(int, int, bool);
    void output( ) const;
private:
    DayOfYear date;
    bool parkingEnforcement;
};
```

# Constant Member Functions (2/2)

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

**const modifier must be presented in both function declaration & definition**

```
void f() { // if Holiday::output() is NOT a const member function
  Holiday dragon_boat(6, 6, true); const Holiday new_year(1, 1, true);
  dragon_boat.output();   // ok
  new_year.output();      // compilation error!
}
```

# Inline Member Functions (1/2)

- For non-member functions
  - use keyword *inline* in function declaration and function heading
  - discussed in Chapter 3 already
- In-class function definition
  - place function definition **inside** class definition
    ➔ automatically inline
  - place function definition outside class definition ➔ add **inline** prefix
- Use for very short functions only
- Code actually inserted in place of call
  - eliminate overhead of function call
  - more efficient, but only when short!

# Inline Member Functions (2/2)

```
class rectangle {
public:
     rectangle(double=0, double=0);
     double area() const
    { return length * width; }
     // …
private:
     double length, width;
    // …
};
rectangle::rectangle
(double len, double wid)
: length(len), width(wid) { }
```

```
class rectangle {
public:
     rectangle(double=0, double=0);
     double area() const;
     // …
private:
     double length, width;
    // …
};
rectangle::rectangle
(double len, double wid)
: length(len), width(wid) { }
inline double rectangle::area() const
{  return length * width;  }
```

# Static Data Members (1/3) (Advanced)

- A static data member
  - – a variable that is part of a class,
  - – yet is NOT part of an object of that class
- For a static data member
  - – all objects of that class **SHARE ONLY ONE** copy
- Example: useful for “tracking” within a class
  - – how often a member function is called?
  - – how many objects exist at given time?

# Static Data Members (2/3) (Advanced)

```
class Example {
public:
  static int sv1;    // static data member
  void inc_val() { ++val; }
  void inc_sv2() { ++sv2; }
  void print_val() const { cout << val << endl; }
  void print_sv2() const { cout << sv2 << endl; }
  Example(int v=0) : val(v) {  }
private:
  int val;               // common data member
  static int sv2;    // static data member
};
// static data members MUST be uniquely defined (like global variables)
int Example::sv1 = 100;  // NO static at the beginning
int Example::sv2 =200;  //  definition required even if sv2 is private
```

# Static Data Members (3/3) (Advanced)

```
int main () {
  Example a, b(10);
  a.inc_val();        // result: a.val = 0
  b.print_val();      // 10 will be printed out!
  a.inc_sv2();        // result: Example::sv2 = 201
  b.print_sv2();      // 201 will be printed out!

  cout << Example::sv1 << endl; // ok! sv1 is a PUBLIC data member

  cout << Example::sv2 << endl; // error! sv2 is a private data member

  return 0;
}
```

# Static Member Functions (1/3) (Advanced)

- A static member function
  - needs access to members of a class,
  - yet doesn’t need to be invoked by a particular object
- A static member function can NOT
  - access non-static data members
  - invoke non-static functions!
- A static member function can be called outside class
  - direct call without referring to an object
  - through an object (just like non-static member functions)

Juinn-Dar Huang jdhuang@mail.nctu.edu.tw

Constructors & Other Tools

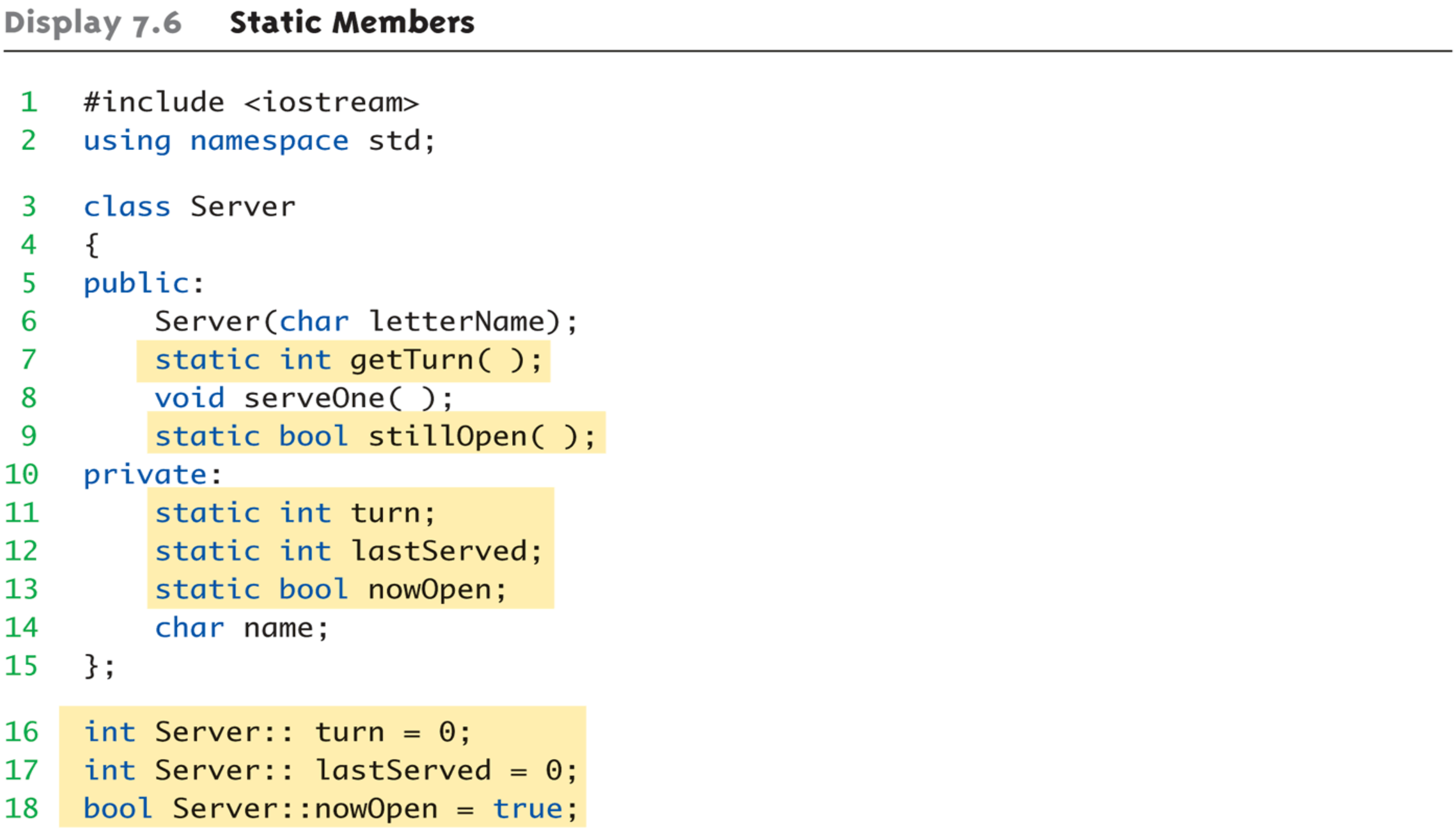
# Static Member Functions (2/3) (Advanced)

```
class Example {
public:
     Example(int v=0) :val(v) { }
     void print_val() const { cout << val << endl; }
     static void s_inc_print_sv();    // static member function
private:
    int val;
    static int sv;                    // static data member
    static void s_inc_sv();   // static member function
};

int Example::sv = 100;    // NO static at the beginning
```

# Static Member Functions (3/3) (Advanced)

```
void Example::s_inc_print_sv() {   // NO static at the beginning
  s_inc_sv();                      // call a static member function
  cout << sv << endl;              // access a static member function
  // cout << val << endl;          // error! cannot access non-static data members!
  // print_val();                  // error! cannot invoke non-static member functions!
}
void Example::s_inc_sv() {  ++sv; }  // NO static at the beginning
int main () {
    Example a;
    a.s_inc_print_sv();              // ok, print 101 out!
    Example::s_inc_print_sv();       // ok, print 102 out!
    a.print_val();                   // ok, print 0 out!
    Example::print_val();            // error! print_val() is not static
    Example::s_inc_sv();             // error! s_inc_sv() is private
    return 0;
}
```

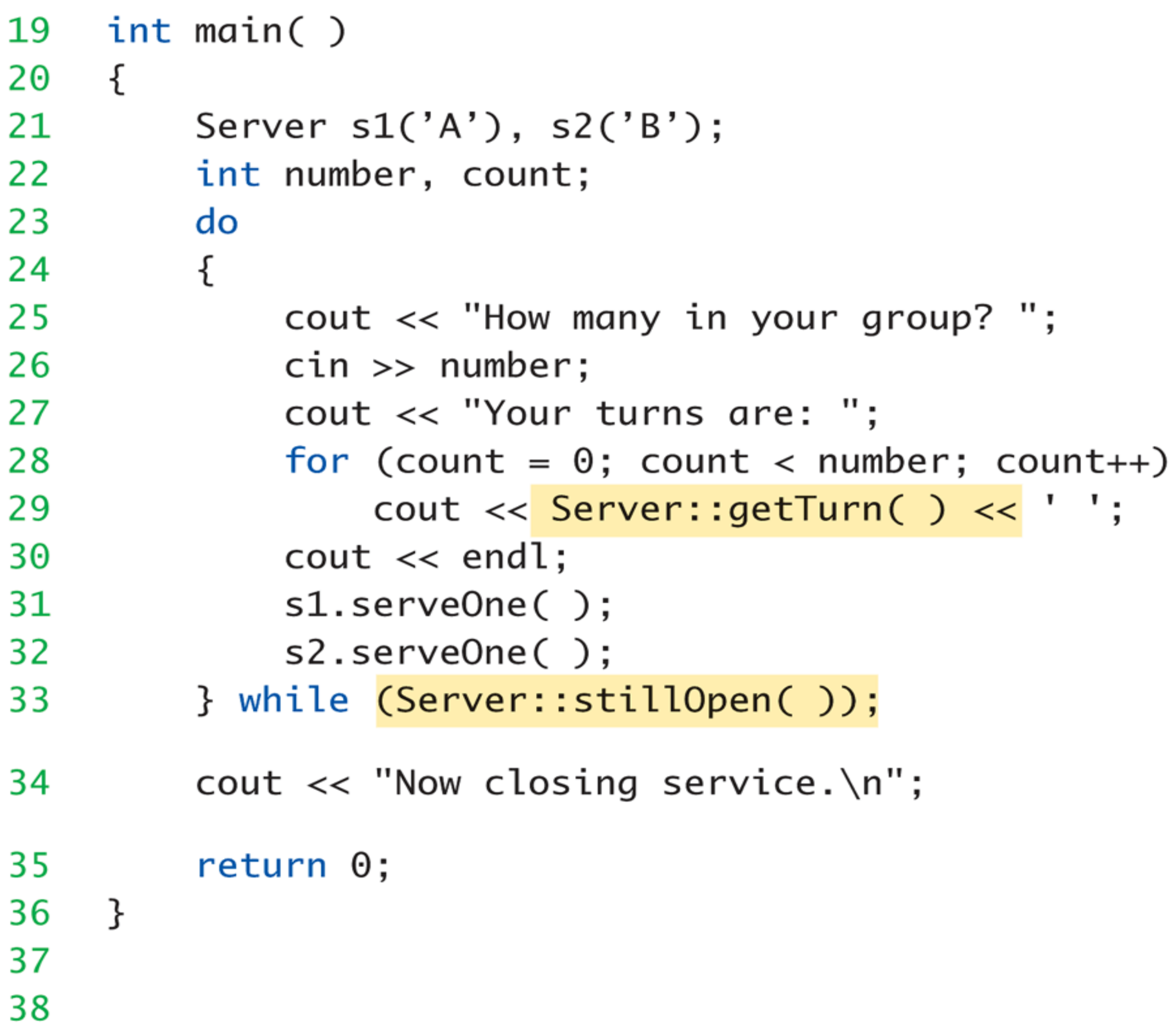
# Example: Static Members (1/4)



**Display 7.6 Static Members**

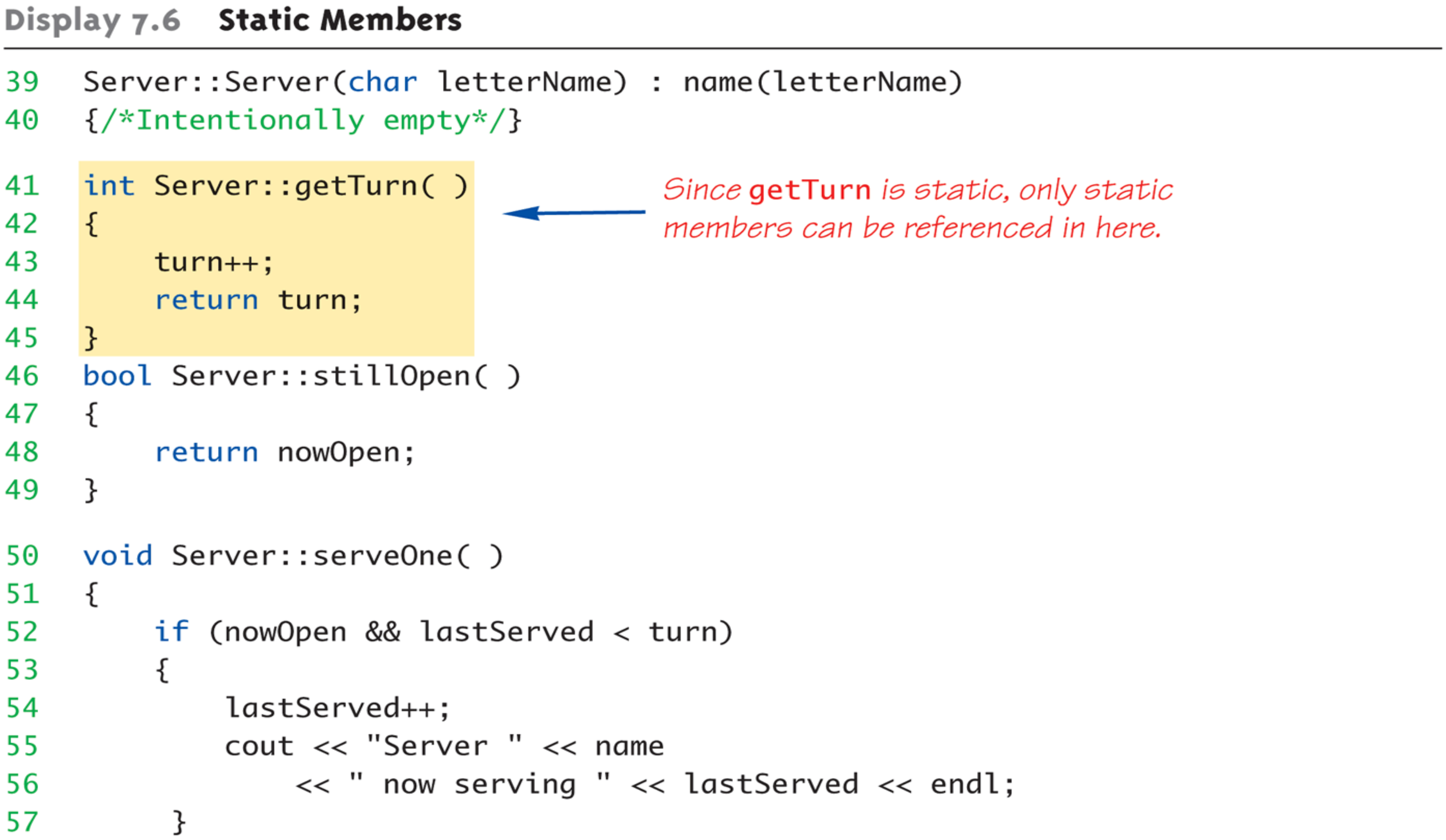
```
#include <iostream>
using namespace std;

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;
```

# Example: Static Members (2/4)



```
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;
}
```

# Example: Static Members (3/4)

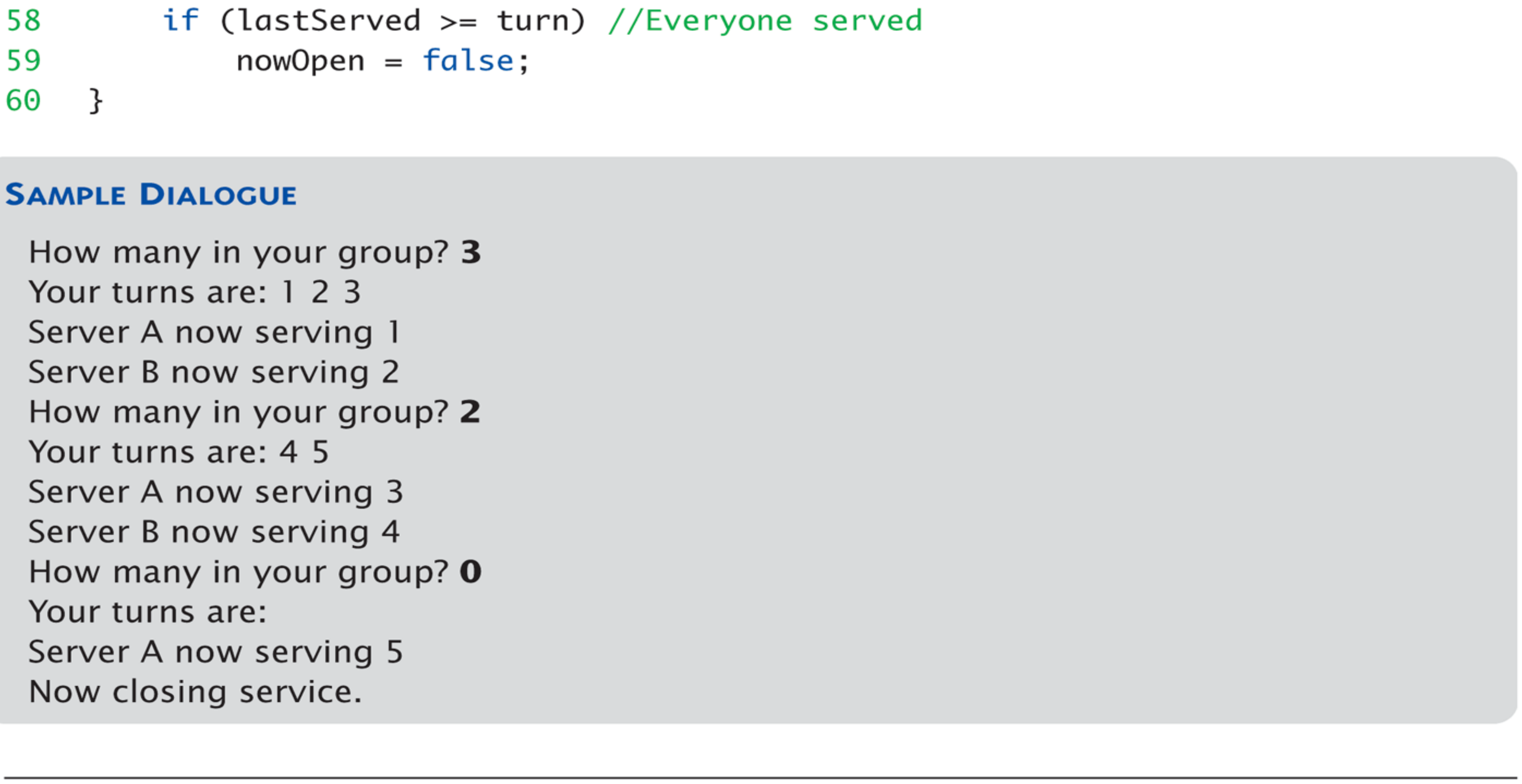


**Display 7.6 Static Members**

```
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.*

# Example: Static Members (4/4)



```
    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.
```

# Summary (1/2)

- Constructors (ctors)
  - name, timing, definitions, and calls
  - overloaded ctors
  - default ctor
  - copy ctor
- Class objects as data members
  - initialized in member initializer list
- Constant member functions
  - guarantee not to alter non-static data members of calling object
- Inline member functions
  - better efficiency, generally for short functions

# Summary (2/2)

- Static data members
  - one copy for an entire class
  - belong to a class, not to an object
  - can be accessed without referring to an object
  - need to be uniquely defined and initialized
    - like global variables
- Static member functions
  - belong to a class, not to an object
  - can be invoked without referring to an object
  - cannot access non-static data members
  - cannot call non-static member functions