Computer Programing (CP)

Lecture # 12

Topic(s)

- Const (Data Members, Member Functions, Objects)
- Static (Data Members, Member Functions)
- This Pointer

Constants

- In C++, we have
  - Const Data Members
  - Const Member Functions
  - Const Objects

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

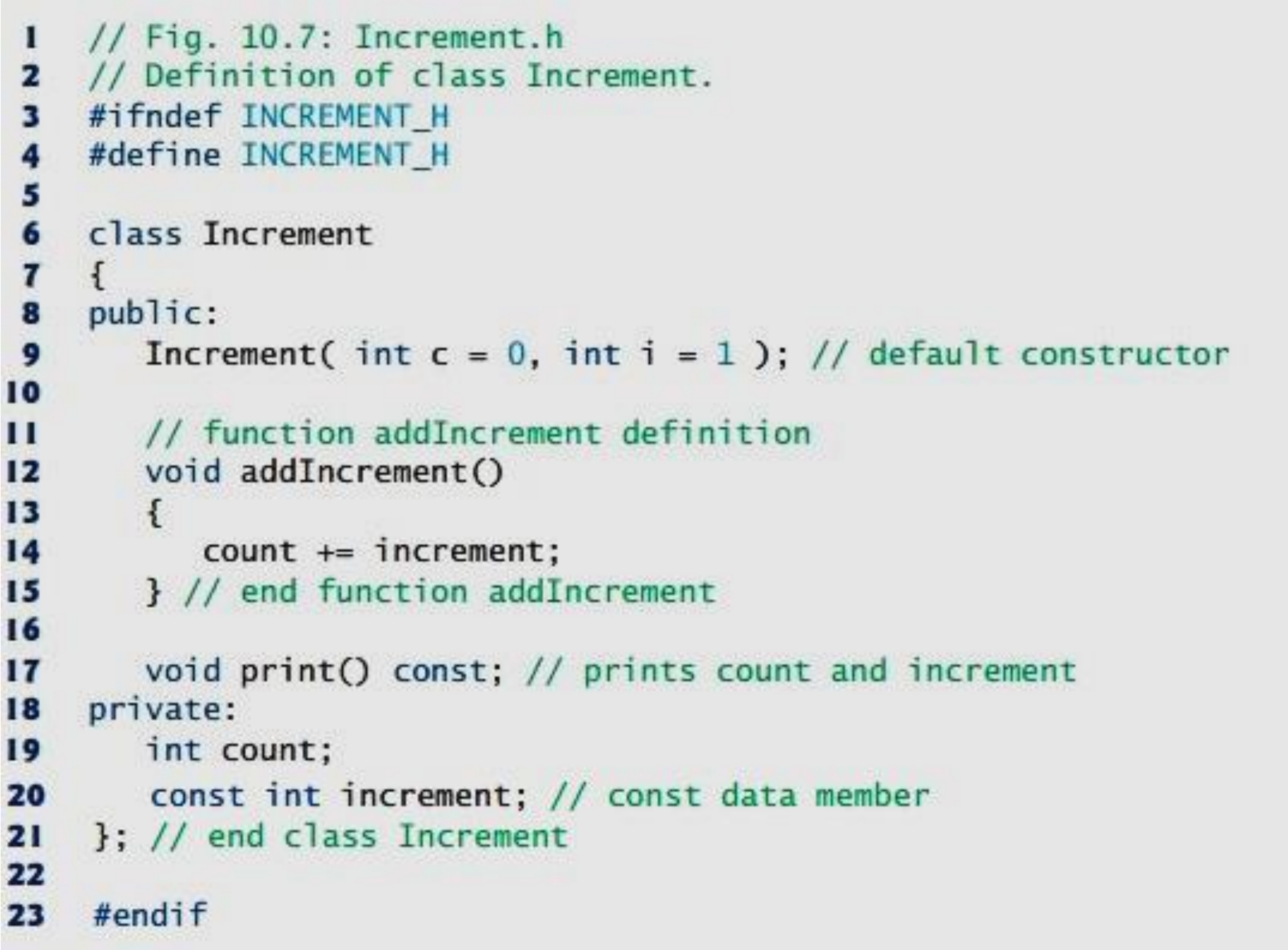
Const Data Members

- Const are those, that value are not changed or remain constant through out the program

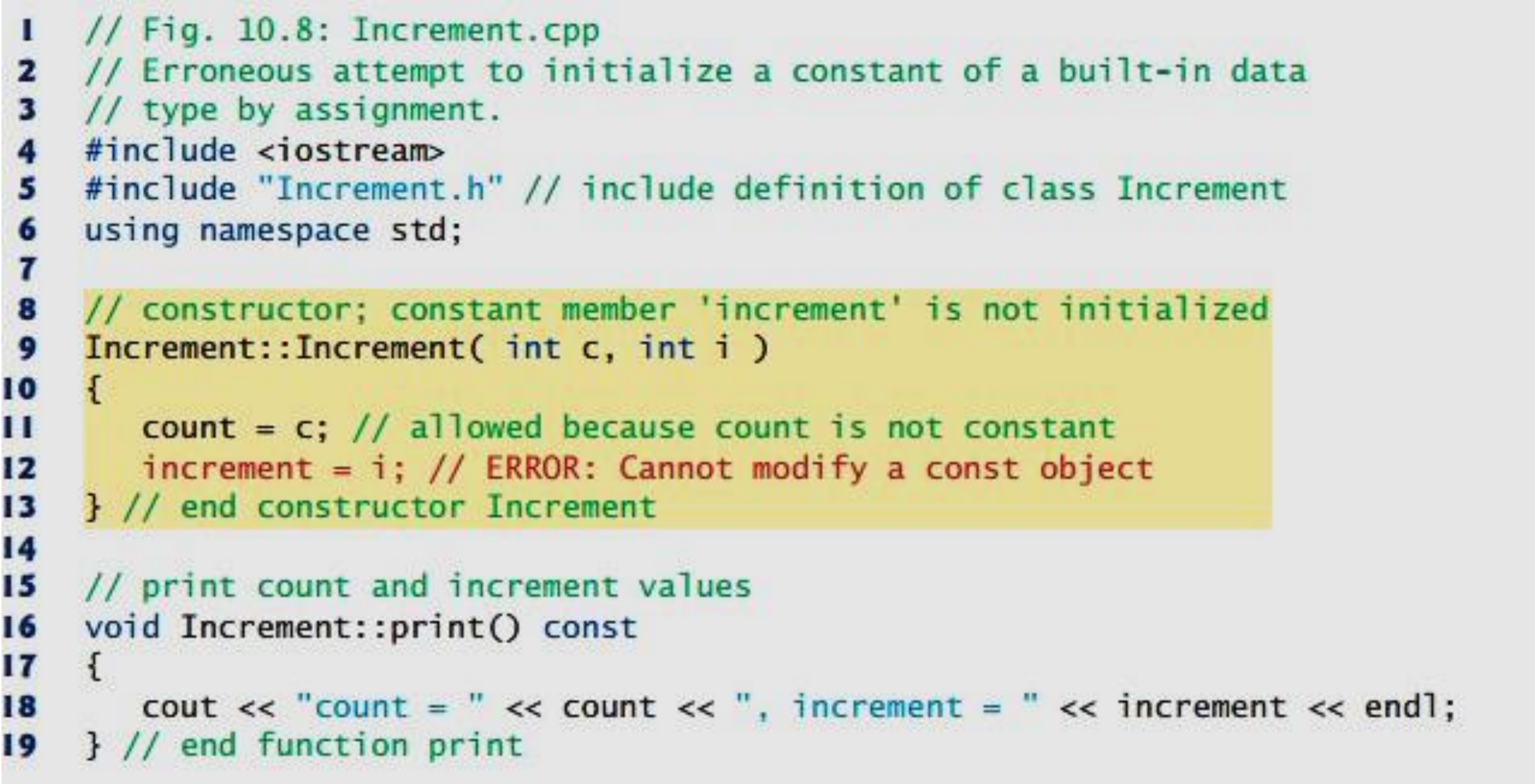
- Intialize at the time of declaration only

- Never update or change value at any stage of program

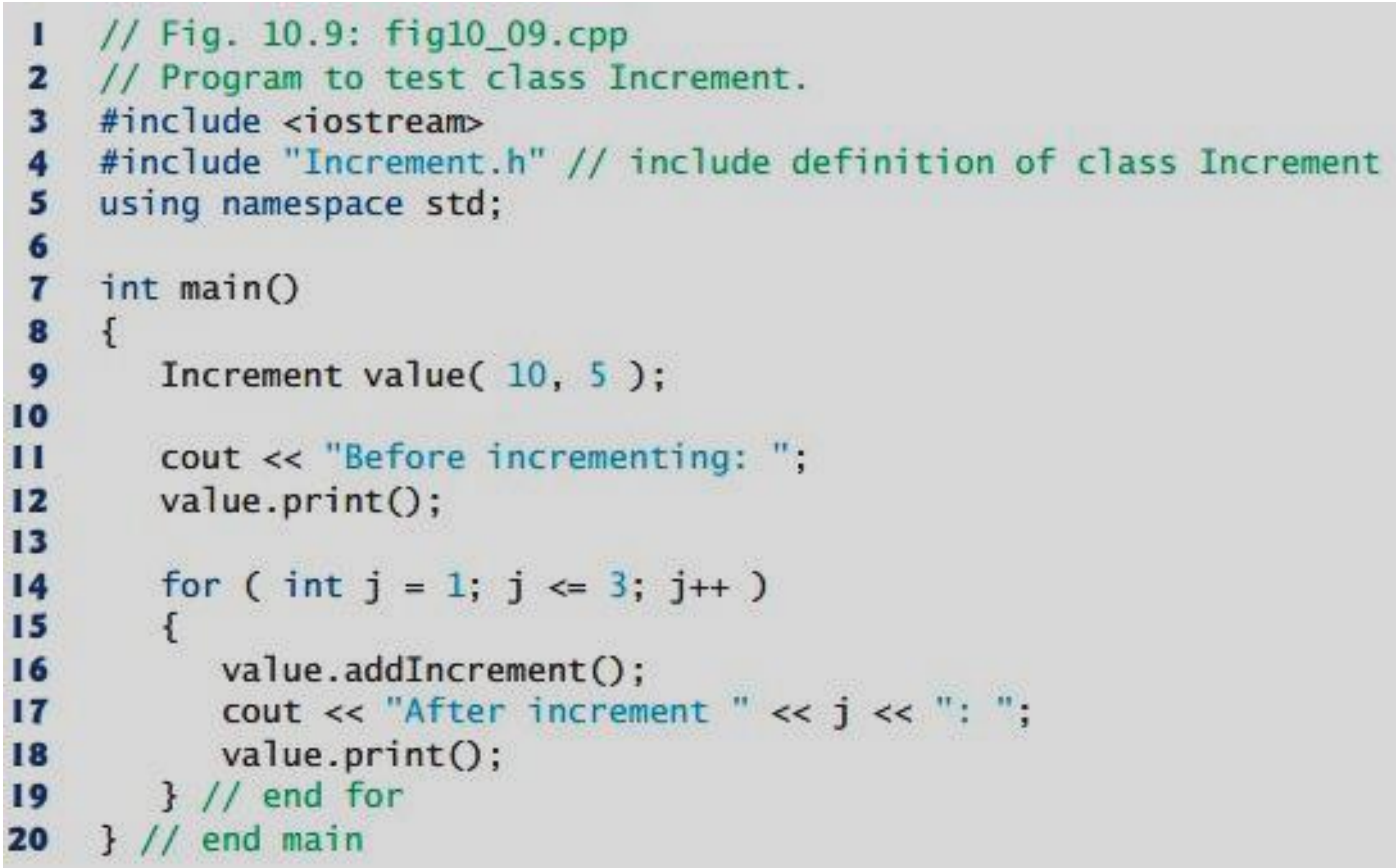
- const int value1 = 5; // copy initialization



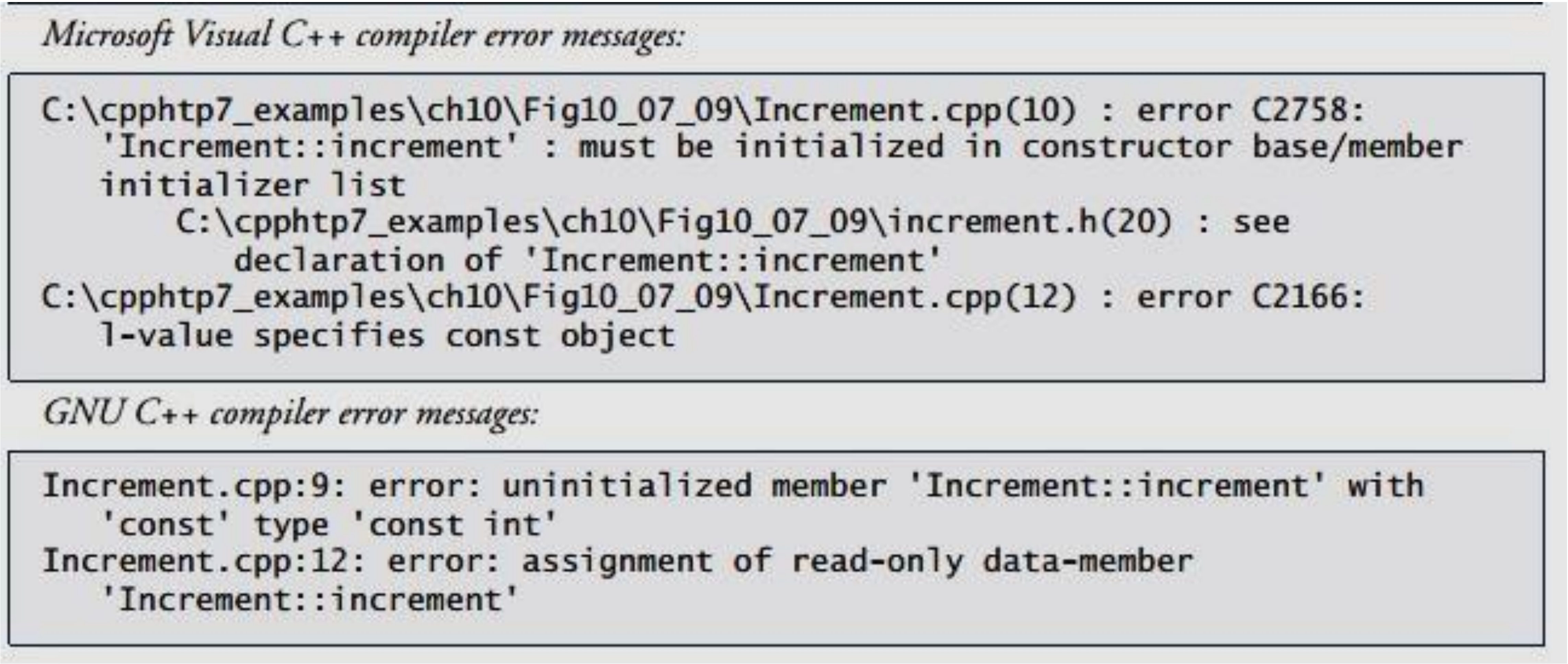
```cpp
// Fig. 10.7: Increment.h
// Definition of class Increment.
#ifndef INCREMENT_H
#define INCREMENT_H

class Increment
{
public:
   Increment( int c = 0, int i = 1 ); // default constructor

   // function addIncrement definition
   void addIncrement()
   {
      count += increment;
   } // end function addIncrement

   void print() const; // prints count and increment
private:
   int count;
   const int increment; // const data member
}; // end class Increment

#endif
```



```
// Fig. 10.8: Increment.cpp
// Erroneous attempt to initialize a constant of a built-in data
// type by assignment.
#include <iostream>
#include "Increment.h" // include definition of class Increment
using namespace std;

// constructor; constant member 'increment' is not initialized
Increment::Increment( int c, int i )
{
   count = c; // allowed because count is not constant
   increment = i; // ERROR: Cannot modify a const object
} // end constructor Increment

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



```cpp
// Fig. 10.9: fig10_09.cpp
// Program to test class Increment.
#include <iostream>
#include "Increment.h" // include definition of class Increment
using namespace std;

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

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

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



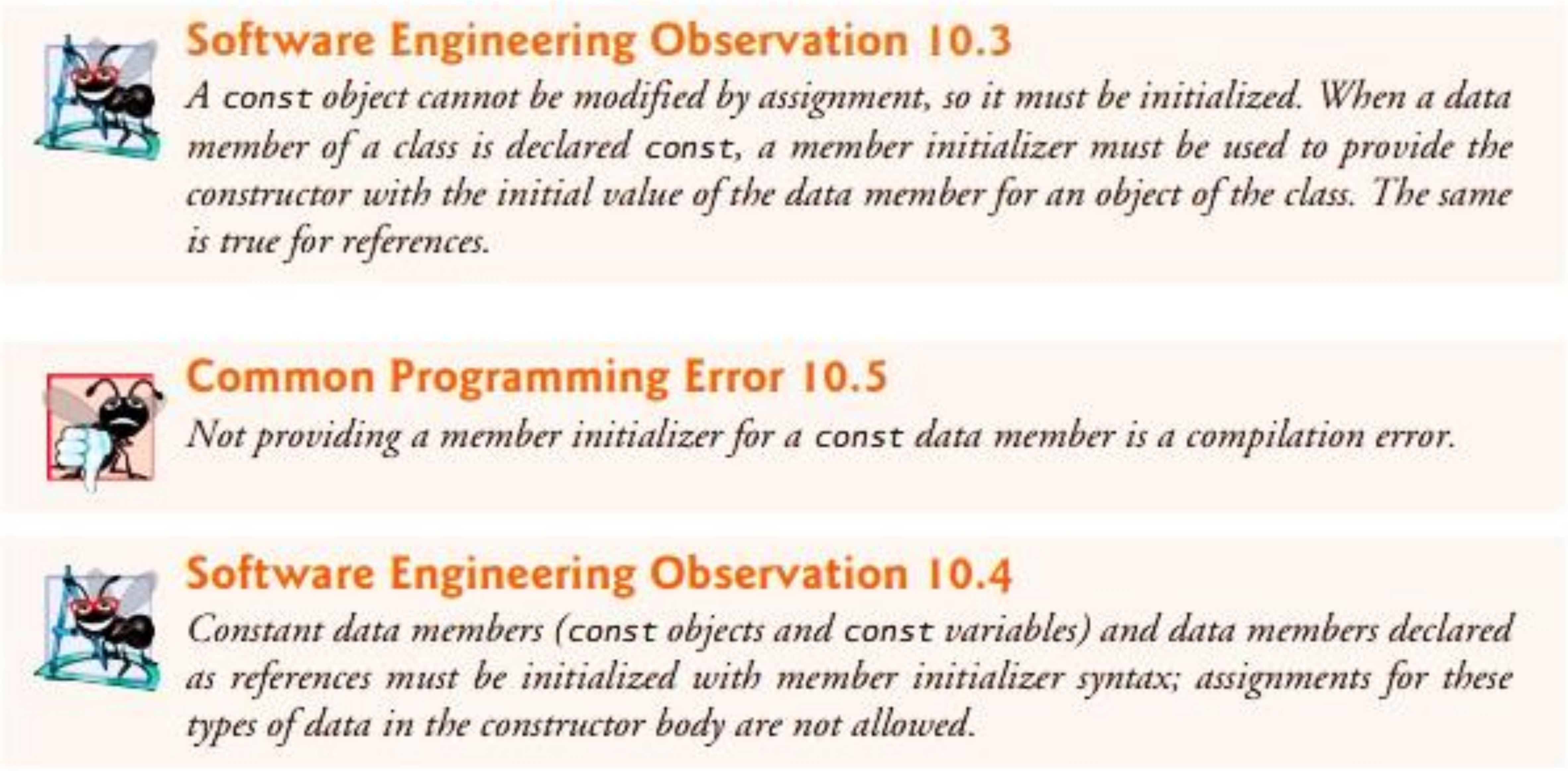
Microsoft Visual C++ compiler error messages:

```
C:\cpphtp7_examples\ch10\Fig10_07_09\Increment.cpp(10) : error C2758:
   'Increment::increment' : must be initialized in constructor base/member
   initializer list
       C:\cpphtp7_examples\ch10\Fig10_07_09\increment.h(20) : see
          declaration of 'Increment::increment'
C:\cpphtp7_examples\ch10\Fig10_07_09\Increment.cpp(12) : error C2166:
   l-value specifies const object
```

GNU C++ compiler error messages:

```
Increment.cpp:9: error: uninitialized member 'Increment::increment' with
   'const' type 'const int'
Increment.cpp:12: error: assignment of read-only data-member
   'Increment::increment'
```

Good Programming Practice(s)



Software Engineering Observation 10.3

A `const` *object cannot be modified by assignment, so it must be initialized. When a data member of a class is declared* `const`*, a member initializer must be used to provide the constructor with the initial value of the data member for an object of the class. The same is true for references.*

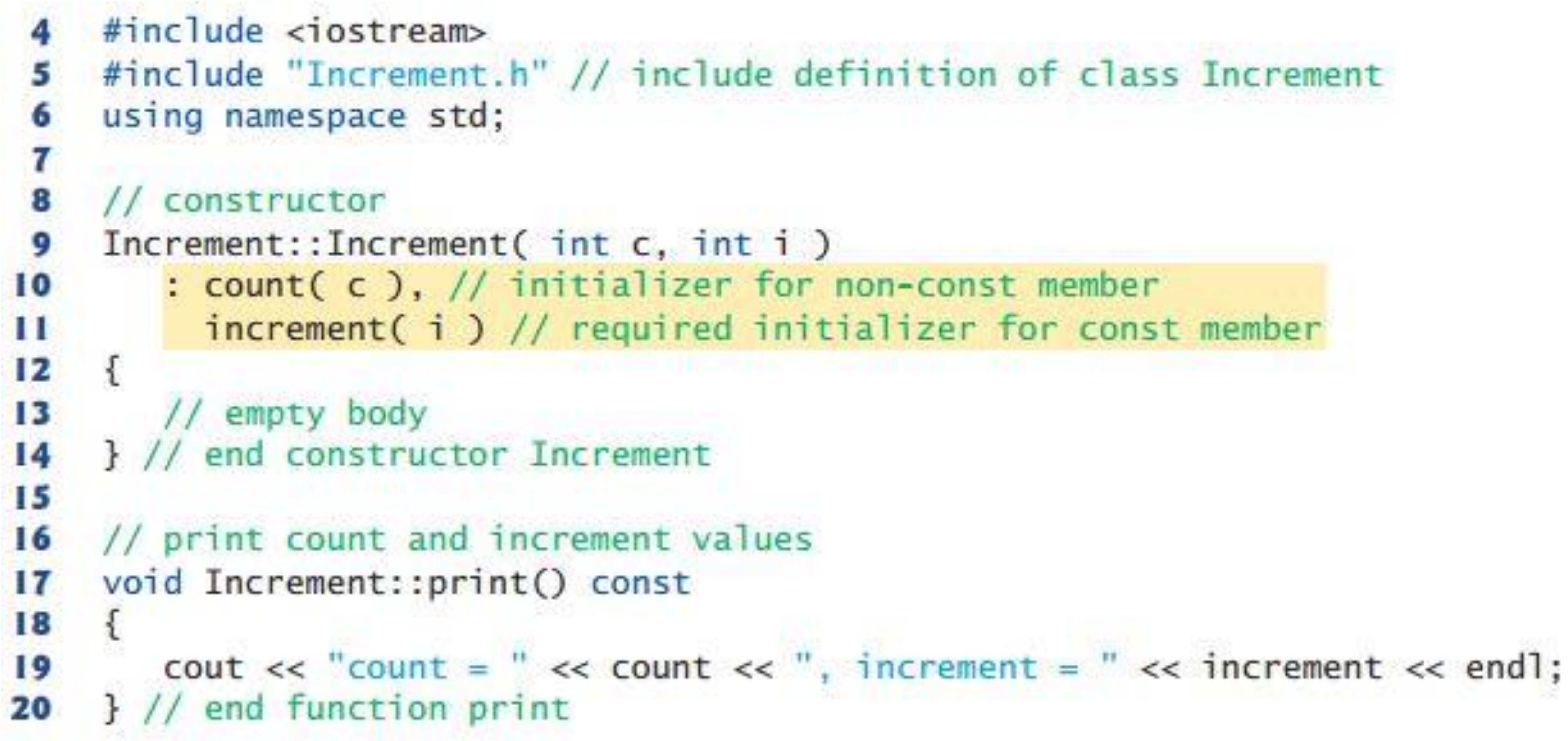
Common Programming Error 10.5

Not providing a member initializer for a `const` *data member is a compilation error.*

Software Engineering Observation 10.4

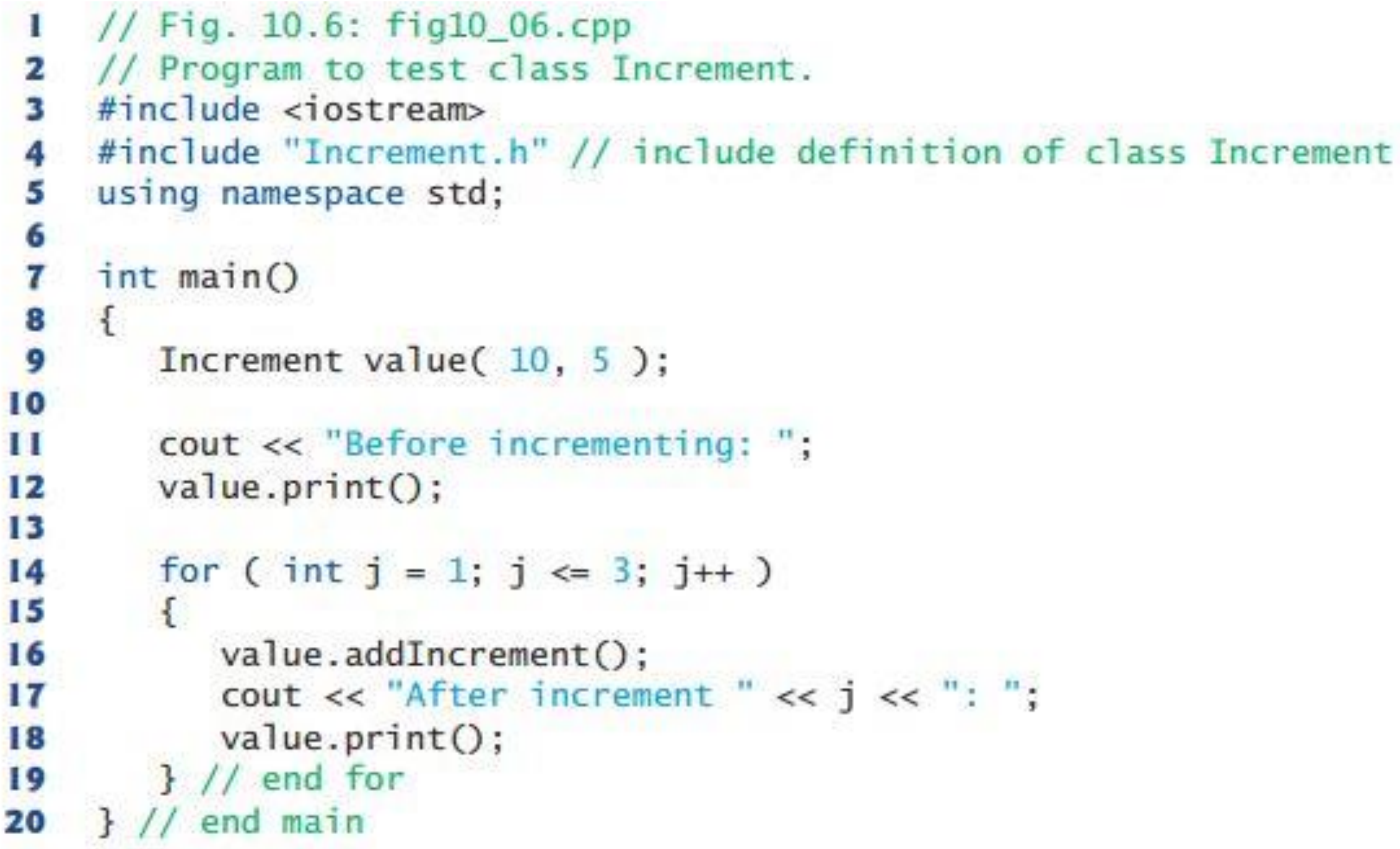
Constant data members (`const` *objects and* `const` *variables) and data members declared as references must be initialized with member initializer syntax; assignments for these types of data in the constructor body are not allowed.*

Correct Initialization

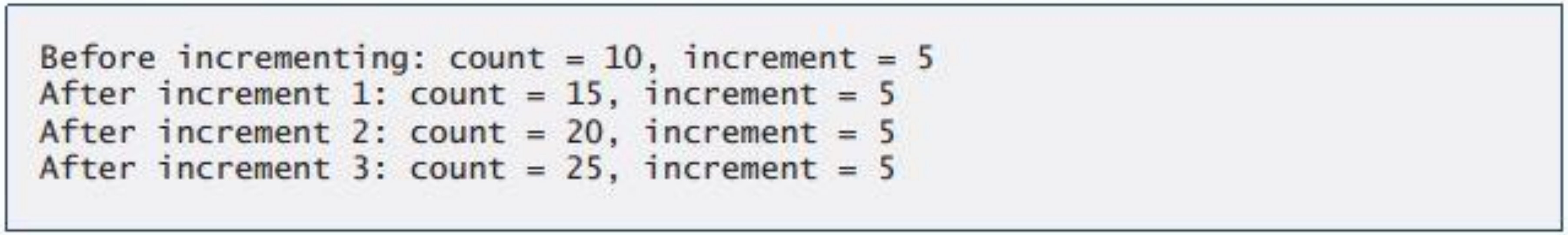


```cpp
#include <iostream>
#include "Increment.h" // include definition of class Increment
using namespace std;

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

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

Output



```cpp
// Fig. 10.6: fig10_06.cpp
// Program to test class Increment.
#include <iostream>
#include "Increment.h" // include definition of class Increment
using namespace std;

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

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

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



```
Before incrementing: count = 10, increment = 5
After increment 1: count = 15, increment = 5
After increment 2: count = 20, increment = 5
After increment 3: count = 25, increment = 5
```

Const Objects & Const Member Function

- The keyword const can be used to specify that an object is not modifiable and that any attempt to modify the object should result in a compilation error.

- Constructors and destructors cannot be declared const.

- A const object must be initialized.

- const data member and reference data members *must* be initialized using member initializers.

Const Objects & Const Member Function

- A member function is specified as const both in its prototype and in its definition.

- Invoking a non-const member function on a const object is a compilation error

- An attempt by a const member function to modify an object of its class is a compilation error.

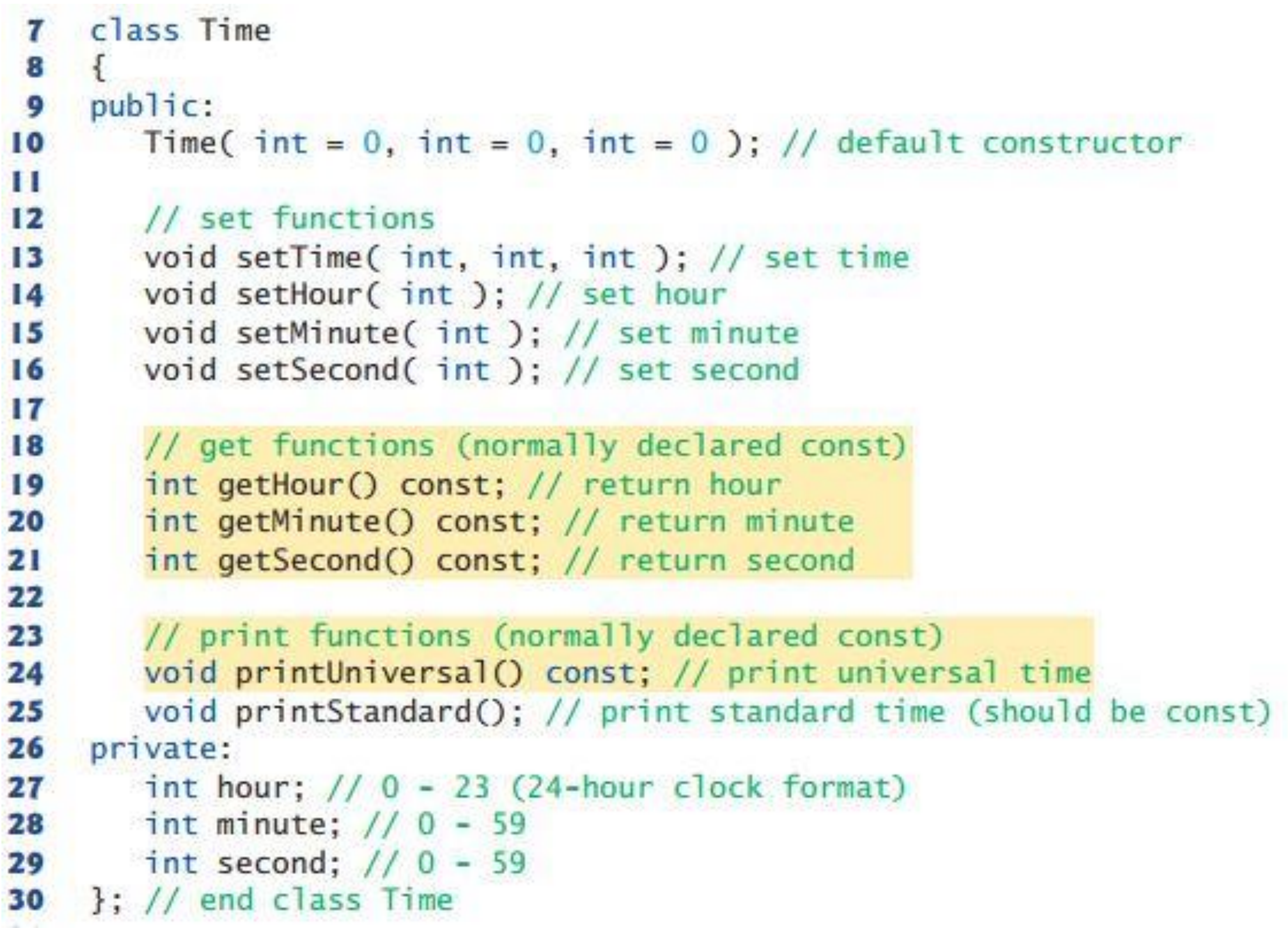
- *A const member function can be overloaded with a non-const version.*

Const Objects & Const Member Function

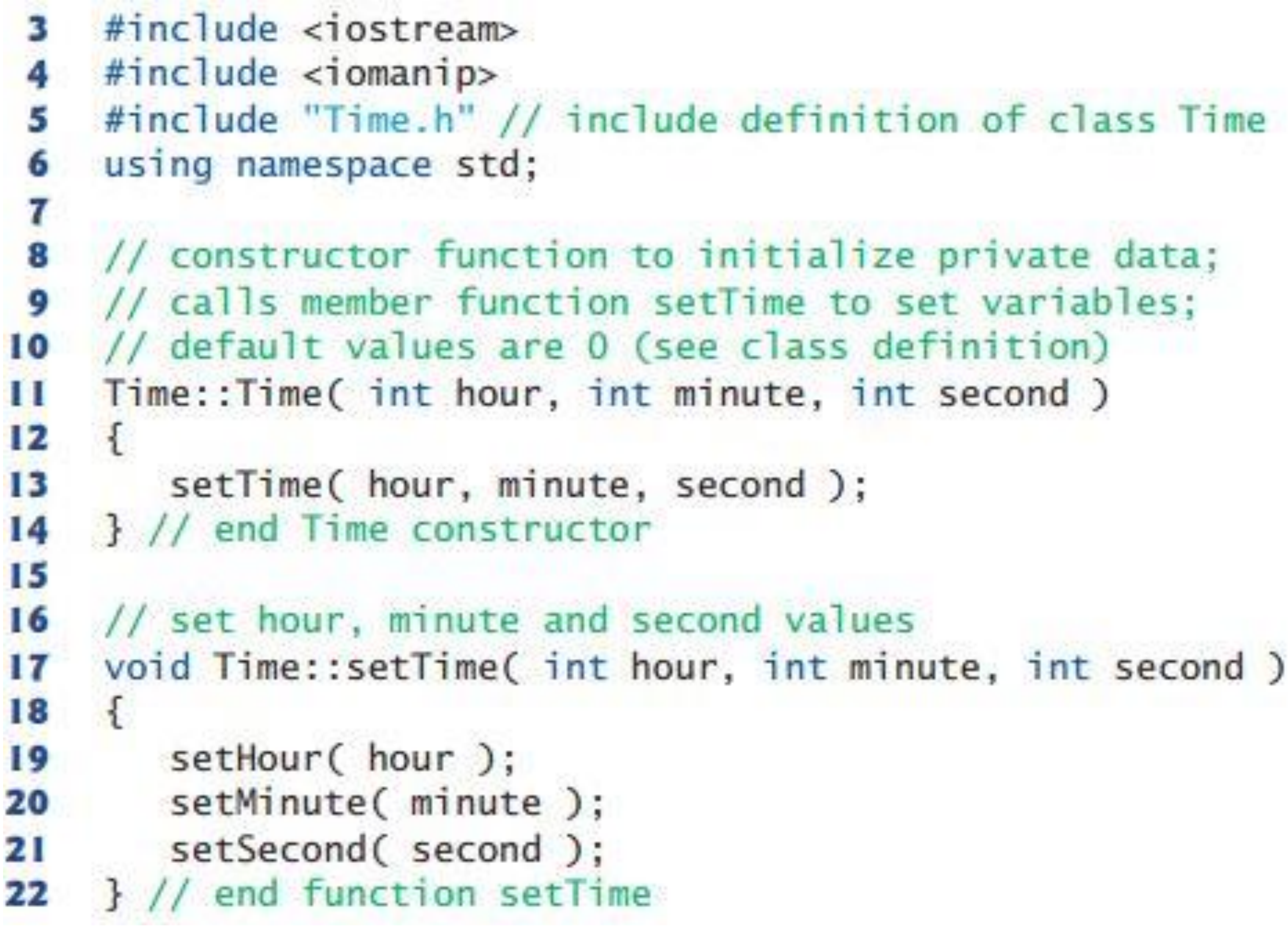
- *A const member function can be overloaded with a non-const version.*
- *The compiler chooses which overloaded member function to use based on the object on which the function is invoked.*
- *If the object is const, the compiler uses the const version.*
- *If the object is not const, the compiler uses the non-const version.*

Declaring & Initializing Const Object

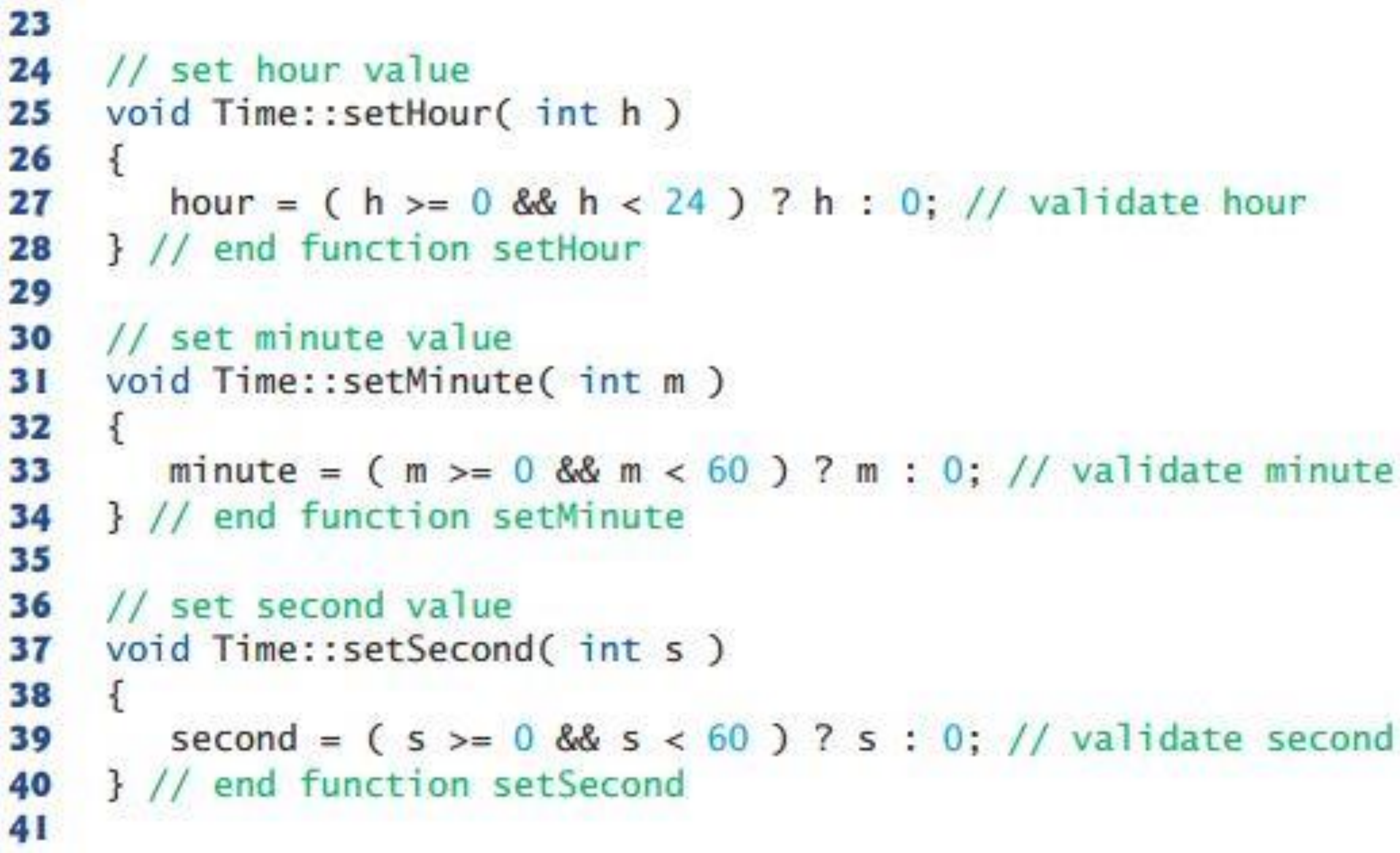
```
const Date date1; // initialize using default constructor
const Date date2(2020, 10, 16); // initialize using parameterized constructor
const Date date3 { 2020, 10, 16 }; // initialize using parameterized constructor (C++11)
```



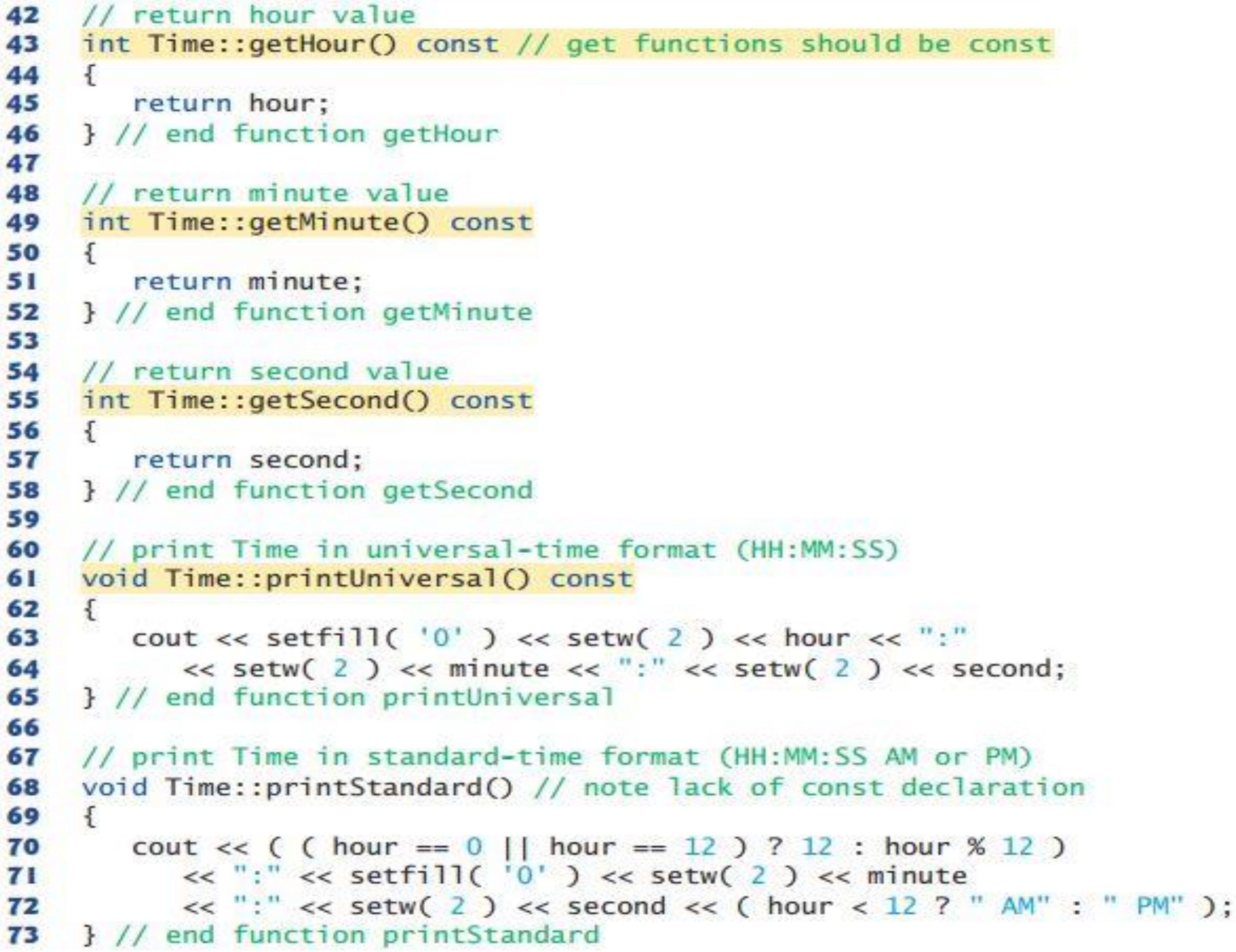
```cpp
class Time
{
public:
   Time( int = 0, int = 0, int = 0 ); // default constructor

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

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

   // print functions (normally declared const)
   void printUniversal() const; // print universal time
   void printStandard(); // print standard time (should be const)
private:
   int hour; // 0 - 23 (24-hour clock format)
   int minute; // 0 - 59
   int second; // 0 - 59
}; // end class Time
```



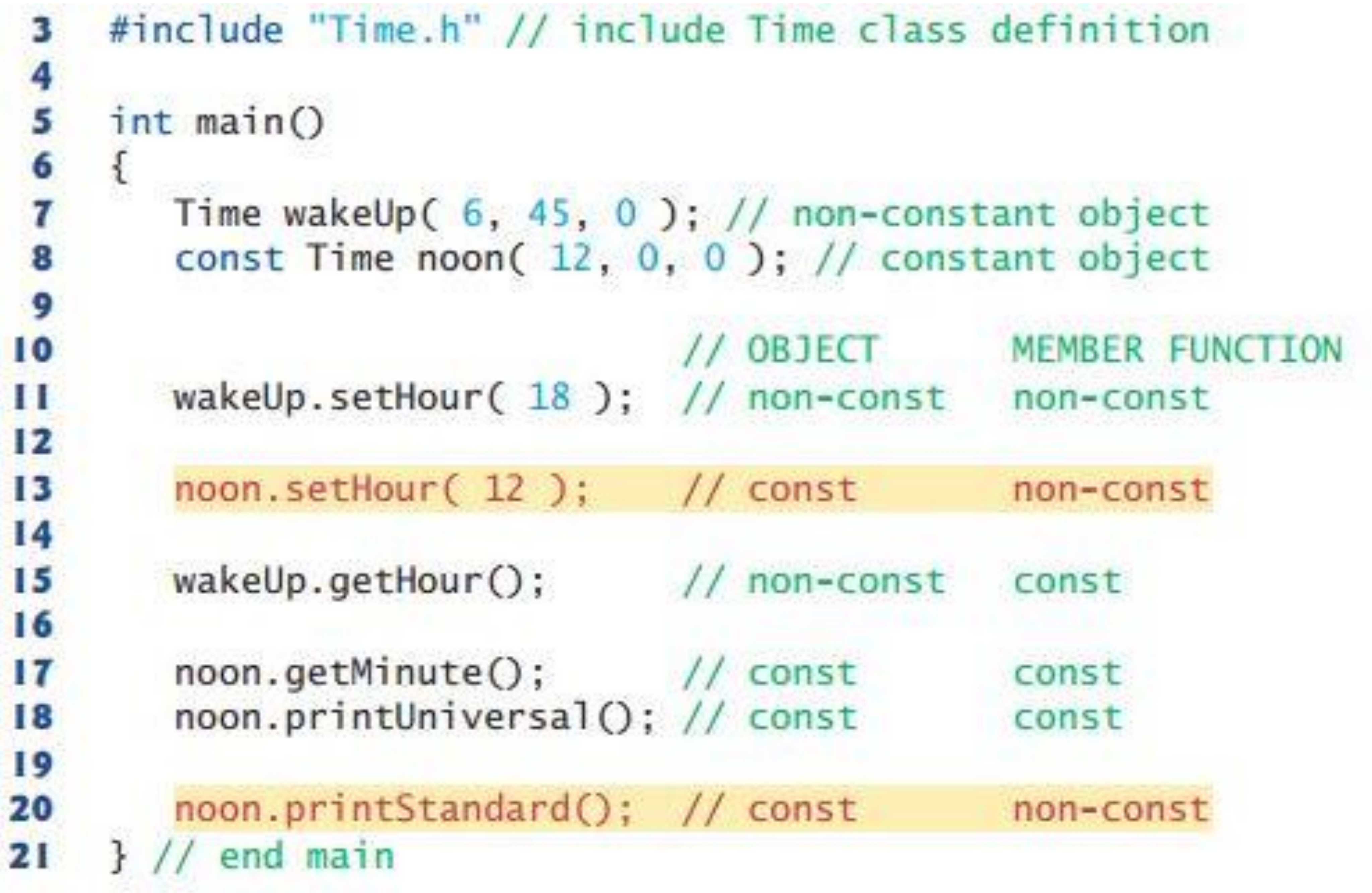
```cpp
#include <iostream>
#include <iomanip>
#include "Time.h" // include definition of class Time
using namespace std;

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

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



```
// set hour value
void Time::setHour( int h )
{
   hour = ( h >= 0 && h < 24 ) ? h : 0; // validate hour
} // end function setHour

// set minute value
void Time::setMinute( int m )
{
   minute = ( m >= 0 && m < 60 ) ? m : 0; // validate minute
} // end function setMinute

// set second value
void Time::setSecond( int s )
{
   second = ( s >= 0 && s < 60 ) ? s : 0; // validate second
} // end function setSecond

```



```
// return hour value
int Time::getHour() const // get functions should be const
{
   return hour;
} // end function getHour

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

// return second value
int Time::getSecond() const
{
   return second;
} // end function getSecond

// print Time in universal-time format (HH:MM:SS)
void Time::printUniversal() const
{
   cout << setfill( '0' ) << setw( 2 ) << hour << ":"
      << setw( 2 ) << minute << ":" << setw( 2 ) << second;
} // end function printUniversal

// print Time in standard-time format (HH:MM:SS AM or PM)
void Time::printStandard() // note lack of const declaration
{
   cout << ( ( hour == 0 || hour == 12 ) ? 12 : hour % 12 )
      << ":" << setfill( '0' ) << setw( 2 ) << minute
      << ":" << setw( 2 ) << second << ( hour < 12 ? " AM" : " PM" );
} // end function printStandard
```



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

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

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

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

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

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

   noon.printStandard();  // const       non-const
} // end main
```

Microsoft Visual C++ compiler error messages:

```
C:\cpphtp7_examples\ch10\Fig10_01_03\fig10_03.cpp(13) : error C2662:
   'Time::setHour' : cannot convert 'this' pointer from 'const Time' to
   'Time &'
        Conversion loses qualifiers
C:\cpphtp7_examples\ch10\Fig10_01_03\fig10_03.cpp(20) : error C2662:
   'Time::printStandard' : cannot convert 'this' pointer from 'const Time' to
   'Time &'
        Conversion loses qualifiers
```

GNU C++ compiler error messages:

```
fig10_03.cpp:13: error: passing 'const Time' as 'this' argument of
   'void Time::setHour(int)' discards qualifiers
fig10_03.cpp:20: error: passing 'const Time' as 'this' argument of
   'void Time::printStandard()' discards qualifiers
```

Static

- Static is a keyword in C++ used to give special characteristics to an element.

- Static elements are allocated storage only once in a program lifetime.

- And they have a scope till the program lifetime. Static Keyword can be used with following,

Static Data Member

- When a **data member** is declared as **static**, only one copy of the **data** is maintained for all objects of the class.
- **Static data members** are not part of objects of a given class type.
- They are shared by all instances of the class
- They do not belong to any particular instance of a class

Static Data Member

- Keyword static is used to make a data member static
- Static data member is declared inside the class
- But they are defined outside the class

```
class ClassName{
…
static DataType VariableName;
};
DataType ClassName::VariableName;
```

Initializing Static Data Member

- Static data members should be initialized once at file scope
- They are initialized at the time of definition

```
class Student{
private:
        static int noOfStudents;
public:
  …
};
int Student::noOfStudents = 0;
/*private static member cannot be accessed outside the class except for
initialization*/
```

Initializing Static Data Member

- If static data members are not explicitly initialized at the time of definition then they are initialized to 0

```
int Student::noOfStudents;
```

is equivalent to

```
int Student::noOfStudents=0;
```

Accessing Static Data Member

```
class Student{
public:
        static int noOfStudents;
};
int Student::noOfStudents;

int main(){
        Student aStudent;
        aStudent.noOfStudents = 1;
        Student::noOfStudents = 1;
}
```

- To access a static data member there are two ways
  - Access like a normal data member
  - Access using a scope resolution operator ‘::’

Life of Static Data Member

```
class Student{
public:
        static int noOfStudents;
};
int Student::noOfStudents;

int main(){
    {
        Student aStudent;
        aStudent.noOfStudents = 1;
    }
Student::noOfStudents = 1;
}
```

- They are created even when there is no object of a class
- They remain in memory even when all objects of a class are destroyed
- They can be used to store information that is required by all objects, like global variables

Static Member Function

- Just like the static data members or static variables inside the class, static member functions also does not depend on object of class.

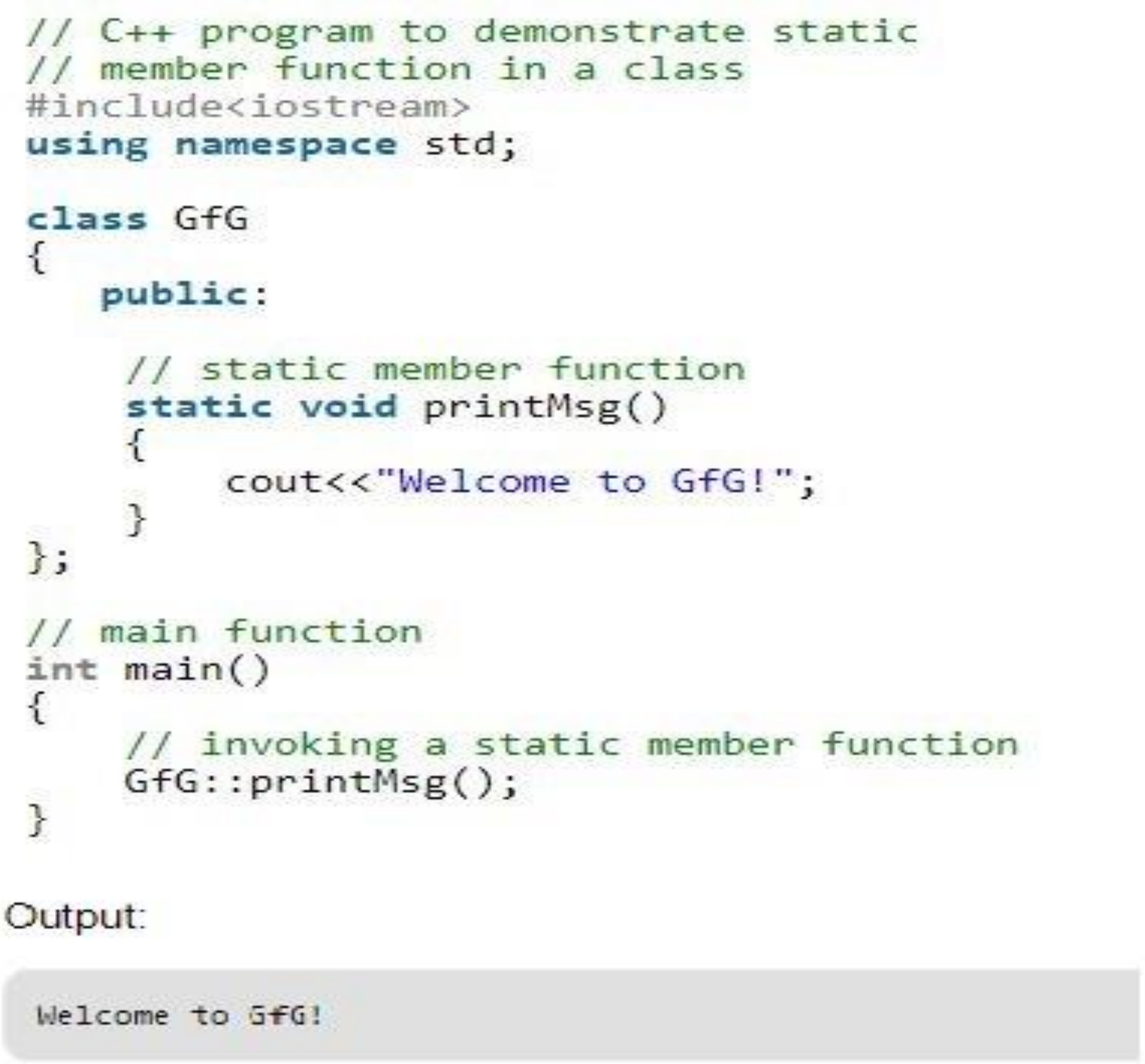
- We are allowed to invoke a static member function using the object and the (.) operator

- but it is recommended to invoke the static members using the class name and the scope resolution (::) operator.

Static Member Function

- Static member functions are allowed to access only the static data members or other static member functions

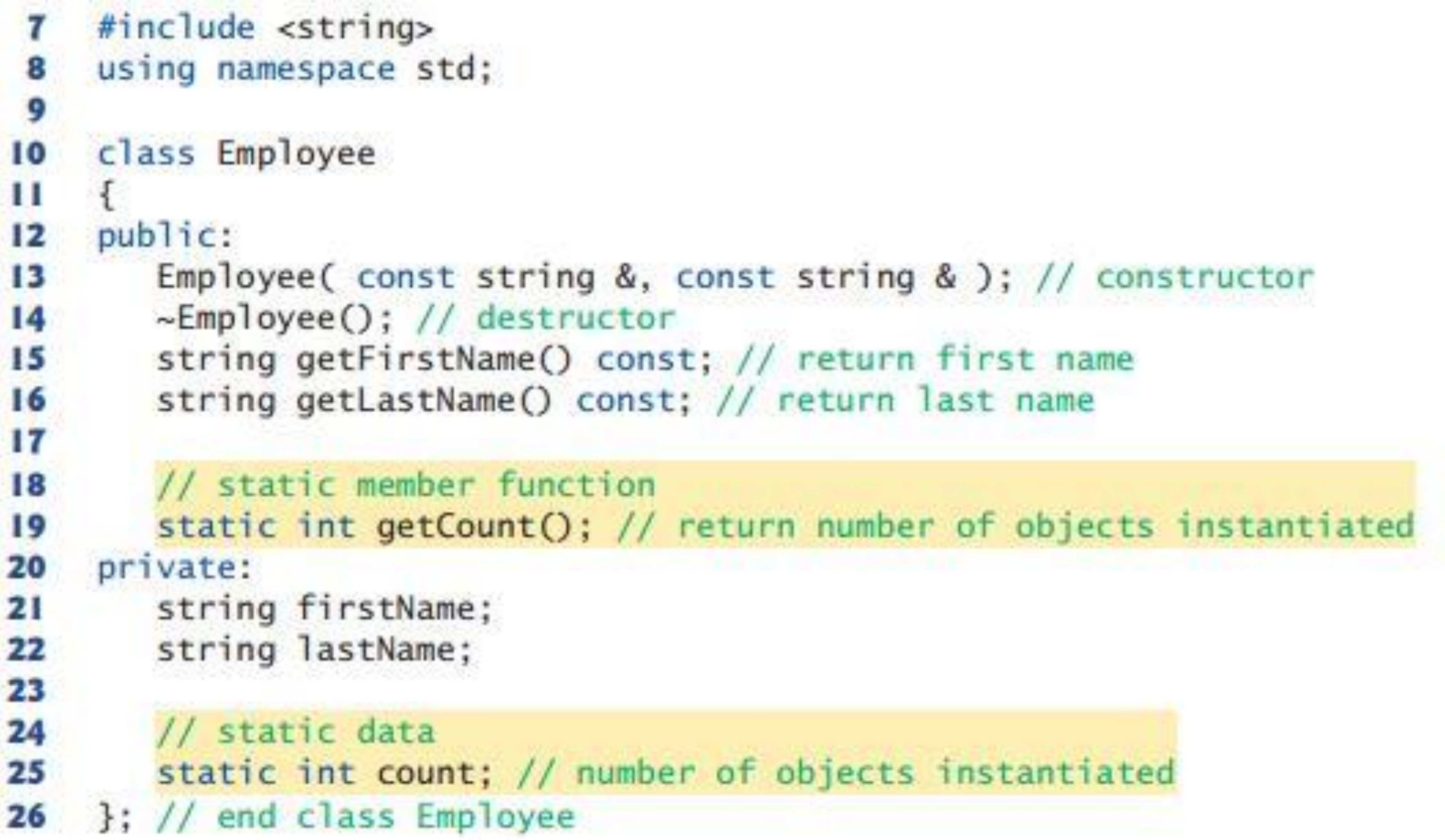
- Static member functions can not access the non-static data members or member functions of the class.



```
// C++ program to demonstrate static
// member function in a class
#include<iostream>
using namespace std;

class GfG
{
   public:

    // static member function
    static void printMsg()
    {
        cout<<"Welcome to GfG!";
    }
};

// main function
int main()
{
    // invoking a static member function
    GfG::printMsg();
}
```

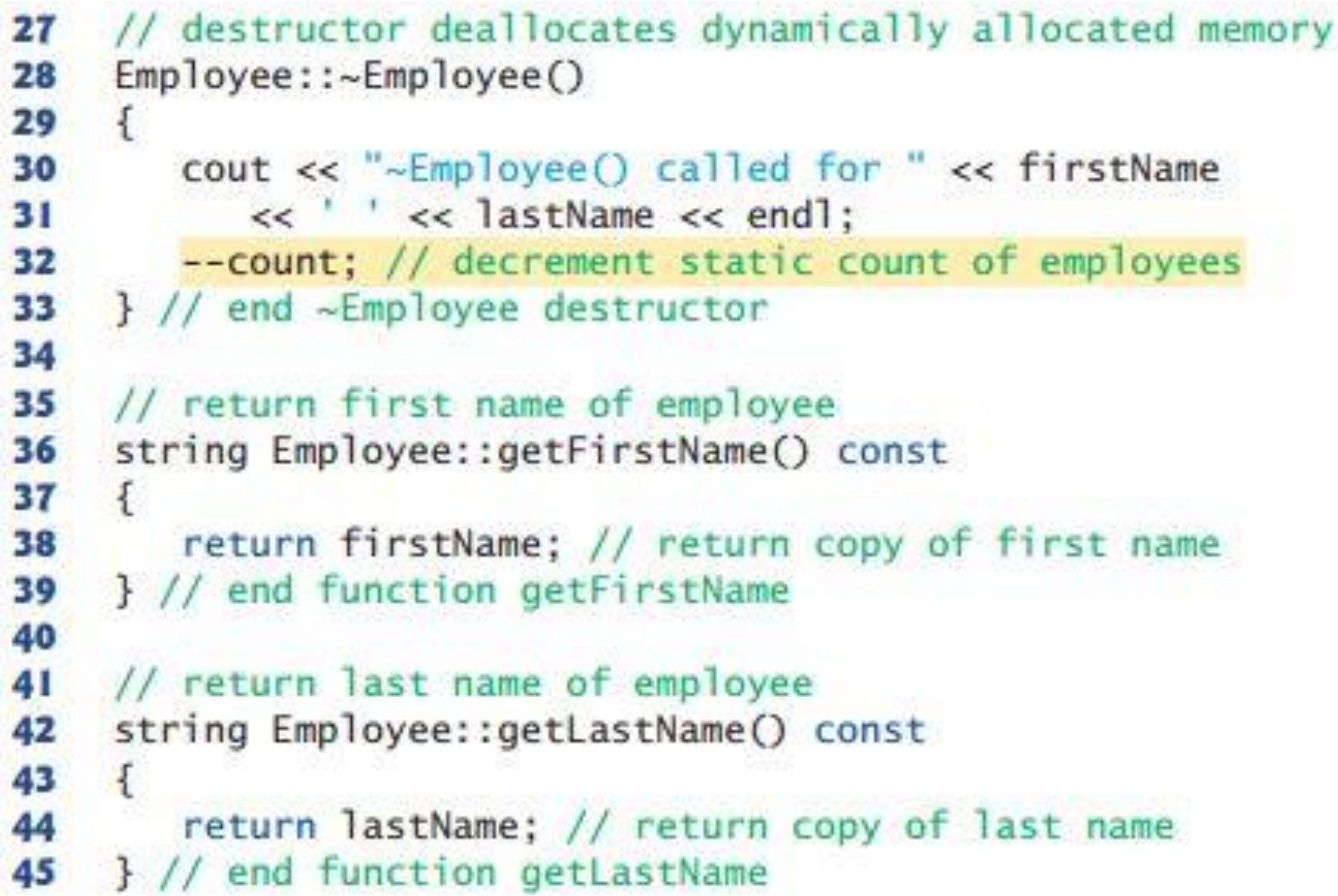
Output:

```
Welcome to GfG!
```

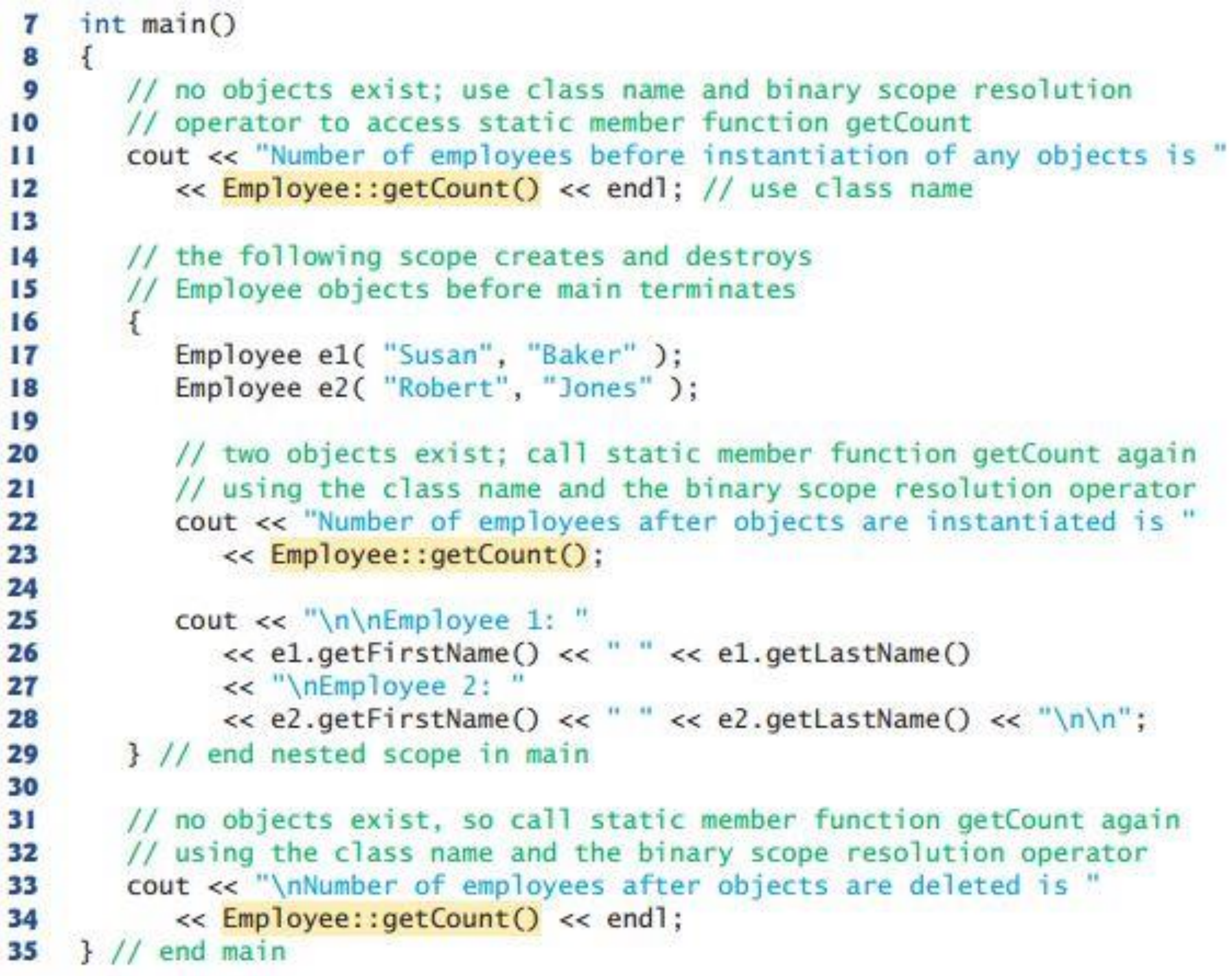


```cpp
#include <string>
using namespace std;

class Employee
{
public:
   Employee( const string &, const string & ); // constructor
   ~Employee(); // destructor
   string getFirstName() const; // return first name
   string getLastName() const; // return last name

   // static member function
   static int getCount(); // return number of objects instantiated
private:
   string firstName;
   string lastName;

   // static data
   static int count; // number of objects instantiated
}; // end class Employee
```

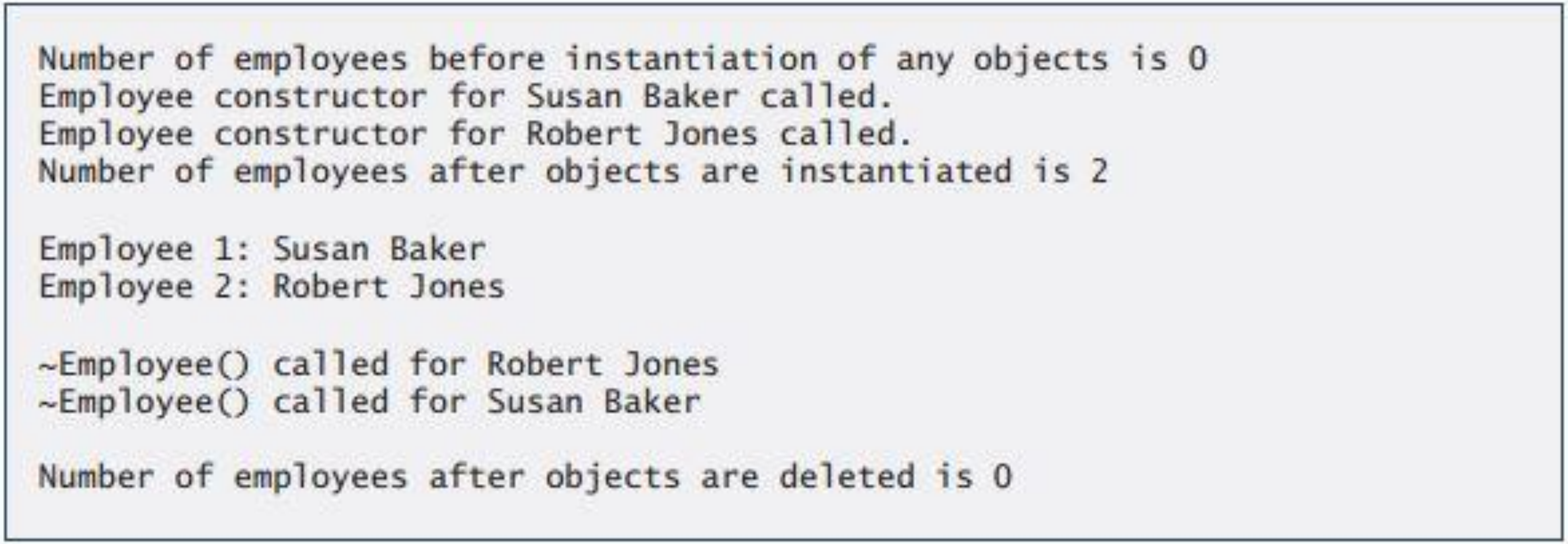
```
// define and initialize static data member at global namespace scope
int Employee::count = 0; // cannot include keyword static

// define static member function that returns number of
// Employee objects instantiated (declared static in Employee.h)
int Employee::getCount()
{
   return count;
} // end static function getCount

// constructor initializes non-static data members and
// increments static data member count
Employee::Employee( const string &first, const string &last )
   : firstName( first ), lastName( last )
{
   ++count; // increment static count of employees
   cout << "Employee constructor for " << firstName
      << ' ' << lastName << " called." << endl;
} // end Employee constructor

```



```cpp
// destructor deallocates dynamically allocated memory
Employee::~Employee()
{
   cout << "~Employee() called for " << firstName
      << ' ' << lastName << endl;
   --count; // decrement static count of employees
} // end ~Employee destructor

// return first name of employee
string Employee::getFirstName() const
{
   return firstName; // return copy of first name
} // end function getFirstName

// return last name of employee
string Employee::getLastName() const
{
   return lastName; // return copy of last name
} // end function getLastName
```



```cpp
int main()
{
   // no objects exist; use class name and binary scope resolution
   // operator to access static member function getCount
   cout << "Number of employees before instantiation of any objects is "
      << Employee::getCount() << endl; // use class name

   // the following scope creates and destroys
   // Employee objects before main terminates
   {
      Employee e1( "Susan", "Baker" );
      Employee e2( "Robert", "Jones" );

      // two objects exist; call static member function getCount again
      // using the class name and the binary scope resolution operator
      cout << "Number of employees after objects are instantiated is "
         << Employee::getCount();

      cout << "\n\nEmployee 1: "
         << e1.getFirstName() << " " << e1.getLastName()
         << "\nEmployee 2: "
         << e2.getFirstName() << " " << e2.getLastName() << "\n\n";
   } // end nested scope in main

   // no objects exist, so call static member function getCount again
   // using the class name and the binary scope resolution operator
   cout << "\nNumber of employees after objects are deleted is "
      << Employee::getCount() << endl;
} // end main
```



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

Employee 1: Susan Baker
Employee 2: Robert Jones

~Employee() called for Robert Jones
~Employee() called for Susan Baker

Number of employees after objects are deleted is 0
```