COS30008

Data Structures and Patterns

Author:
Dr Markus Lumpe
Swinburne University of Technology
Hawthorn Campus

COS30008

Academic Staff:
Dr Pan Zheng
Assoc. Prof. Sim Kwan Yong

Lecture:	Monday 8.30am-10.30am A309
Lab:	Tuesday 1.30pm-3.30pm A302

Assessments:

- Problem Sets (25%)
- Test (25%)
- Final Exam (50%)

COS30008 Assessments

Assessments:

Problem Sets (25%)
Test (25%)
Final Exam (50%)

Subject Aims

- How can a given problem be effectively expressed?
- What are suitable data representations for specifying computational processes?
- What is the impact of data and its representation with respect to time and space consumption?
- What are the reoccurring structural artifacts in software and how can we identify them in order to facilitate problem solving?

Learning Objectives

1. Apply object oriented design and implementation techniques.
2. Interpret the tradeoffs and issues involved in the design, implementation, and application of various data structures with respect to a given problem.
3. Design, implement, and evaluate software solutions using behavioral, creational, and structural software design patterns.
4. Explain the purpose and answer questions about data structures and design patterns that illustrate strengths and weaknesses with respect to resource consumption.
5. Assess the impact of data structures on algorithms.
6. Analyze algorithm designs and perform best-, average-, and worst-case analysis.

Overview

The following gives a tentative list of topics not necessarily in the order in which they will be covered in the subject:

- Introduction
- Sets, Arrays, Indexers, and Iterators
- Basic Data Structures and Patterns
- Abstract Data Types and Data Representation
- One-Dimensional Data Structures
- Hierarchical Data Structures
- Algorithmic Patterns and Problem Solvers

Why?

"Smart data structures and dumb code works a lot better than the other way around."

Eric S. Raymond: The Cathedral and the Bazaar

A brief introduction to C++

Why C++

- We need to know more than just Java.
- C++ is highly efficient and provides a better match to implement low-level software layers like device controllers or networking protocols.
- C++ is being widely used to develop commercial applications and is a the center of operating system and modern game development.
- Memory is tangible in C++ and we can, therefore, study the effects of design decisions on memory management more directly.

Core Properties of Programming Languages

- Programming languages provide us with a framework to organize computational complexity in our own minds.
- Programming languages offer us the means by which we communicate our understanding about a computerized problem solution.

What is C++

- C++ is a general-purpose, high-level programming language with low-level features.
- Bjarne Stroustrup developed C++ (C with Classes) in 1983 at Bell Labs as an enhancement to the C programming language.
- A first C++ programming language standard was ratified in 1998 as ISO/IEC 14882:1998. The current extending version is ISO/IEC 14882:2011 (informally known as C++11).

Design Philosophy of C++

- C++ is a hybrid, statically-typed, general-purpose language that is as efficient and portable as C.
- C++ directly supports multiple programming styles like procedural programming, object-oriented programming, or generic programming.
- C++ gives the programmer choice, even if this makes it possible for the programmer to choose incorrectly!
- C++ avoids features that are platform specific or not general purpose, but is itself platform-dependent.
- C++ does not incur overhead for features that are not used.
- C++ functions without an integrated and sophisticated programming environment.

C++ Paradigms

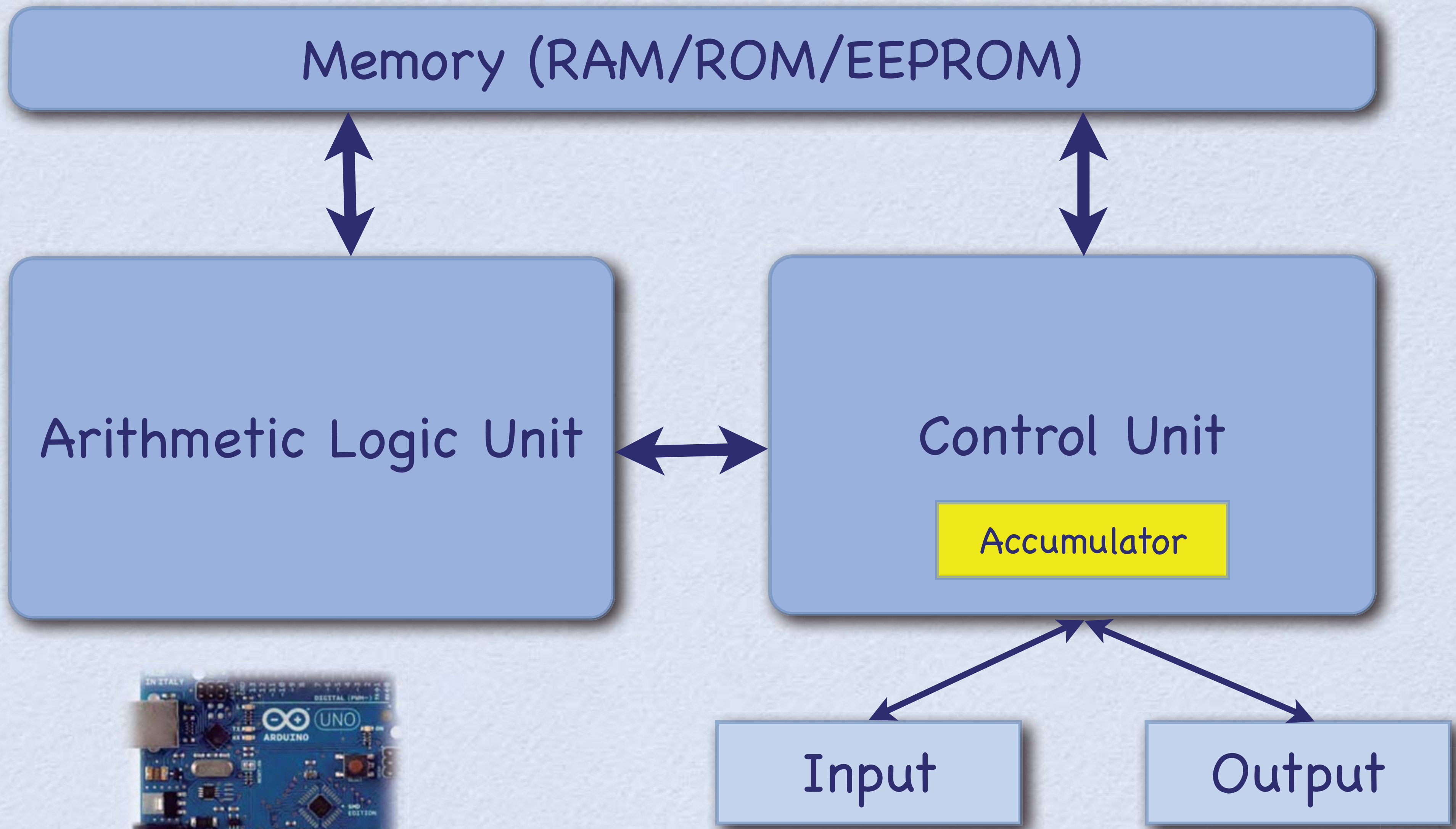
- C++ is a multi-paradigm language.
- C++ provides natural support for
  - the imperative paradigm and
  - the object-oriented paradigm.

- Paradigms must be mixed in any non-trivial project.

Imperative Programming

- This is the oldest style of programming, in which the algorithm for the computation is expressed explicitly in terms of instructions such as assignments, tests, branching and so on.
- Execution of the algorithm requires data values to be held in variables which the program can access and modify.
- Imperative programming corresponds naturally to the earliest, basic and still used model for the architecture of the computer, the von Neumann model.

The von Neumann Architecture



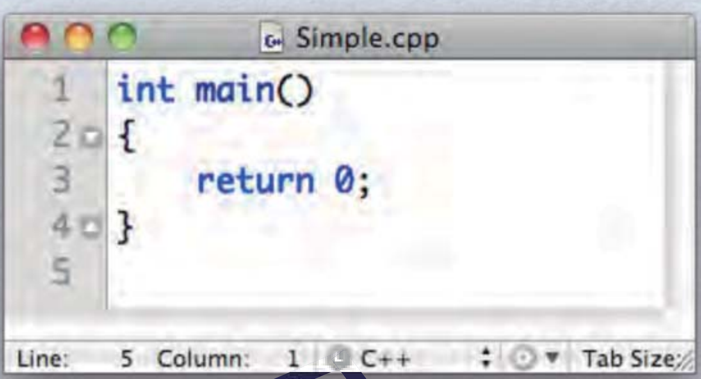
Source: http://arduino.cc/en/Main/ArduinoBoardUno

Object-Oriented Programming

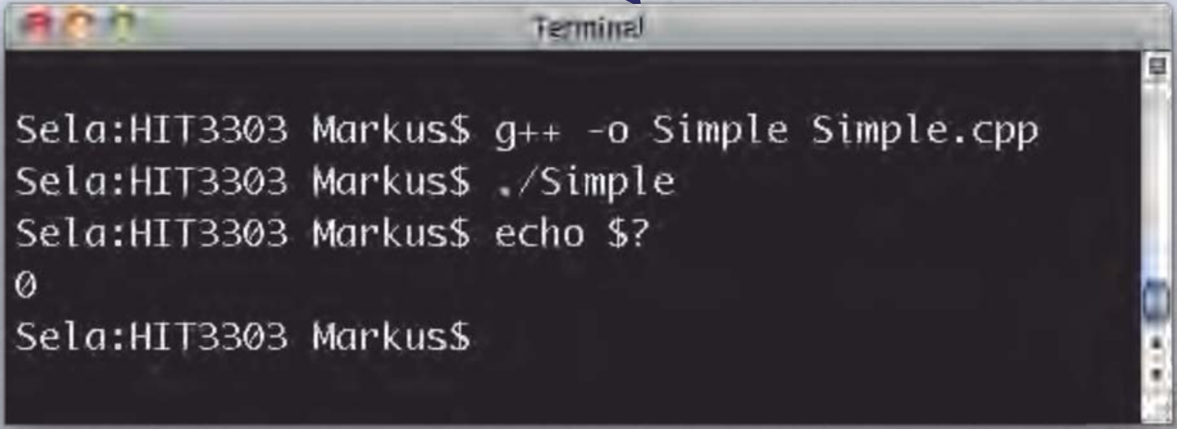
- In general, object-oriented languages are based on the concepts of class and inheritance, which may be compared to those of type and variable respectively in a language like Pascal and C.
- A class describes the characteristics common to all its instances, in a form similar to the record of Pascal (structures in C), and thus defines a set of fields.
- In object-oriented programming, instead of applying global procedures or functions to variables, we invoke the methods associated with the instances (i.e., objects), an action called "message passing."
- The basic concept inheritance is used to derive new classes from exiting ones by modifying or extending the inherited class(es).

The Simplest Possible C++ Program

```
 .text
        .align 1,0x90
.globl _main
_main:
LFB2:
        pushl   %ebp
LCFI0:
        movl    %esp, %ebp
LCFI1:
        subl    $8, %esp
LCFI2:
        movl    $0, %eax
        leave
        ret
LFE2:
        .globl _main.eh
_main.eh = 0
.no_dead_strip _main.eh
        .constructor
        .destructor
        .align 1
        .subsections_via_symbols
```

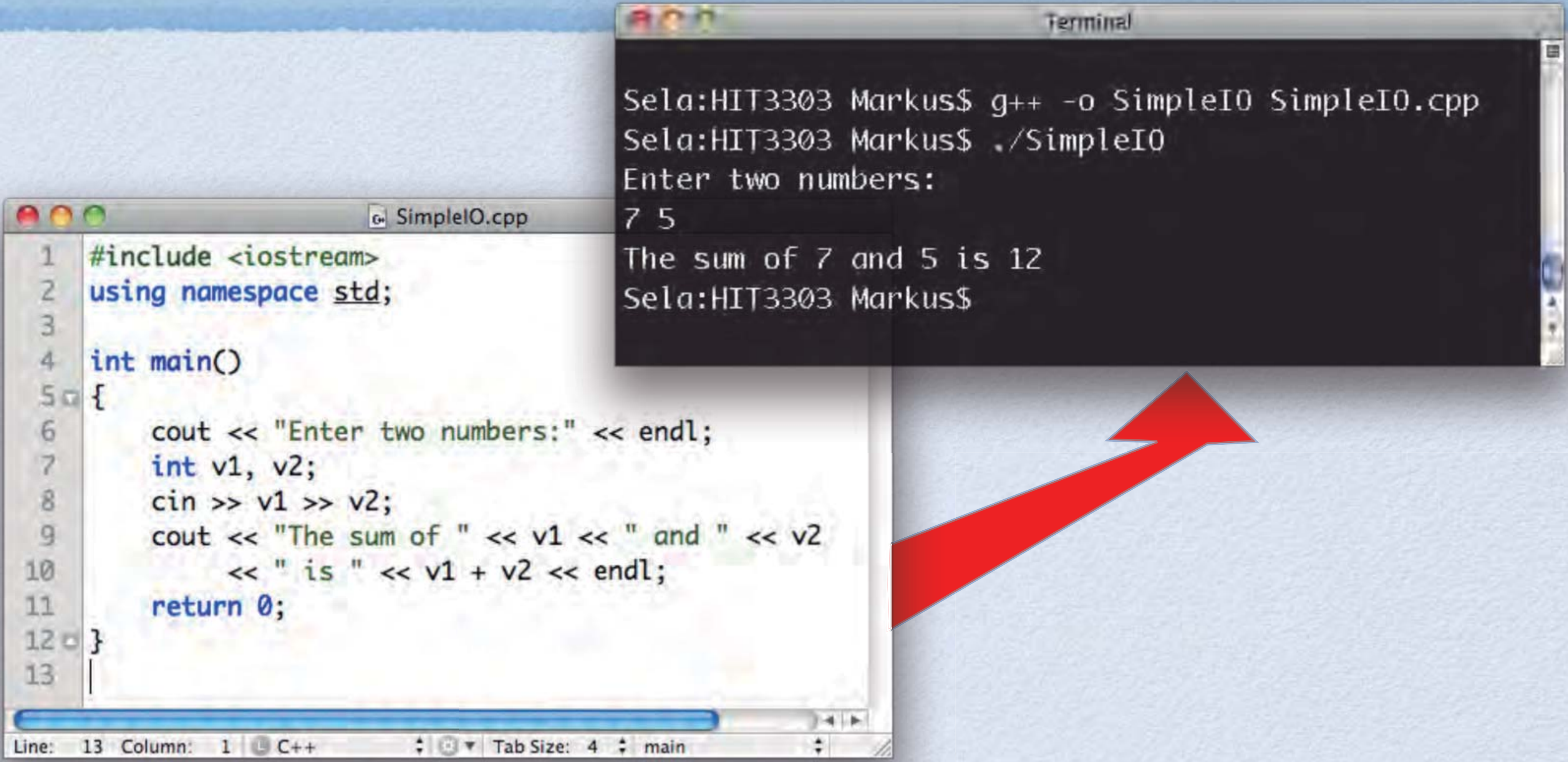


```
int main()
{
    return 0;
}

```



```
Sela:HIT3303 Markus$ g++ -o Simple Simple.cpp
Sela:HIT3303 Markus$ ./Simple
Sela:HIT3303 Markus$ echo $?
0
Sela:HIT3303 Markus$
```

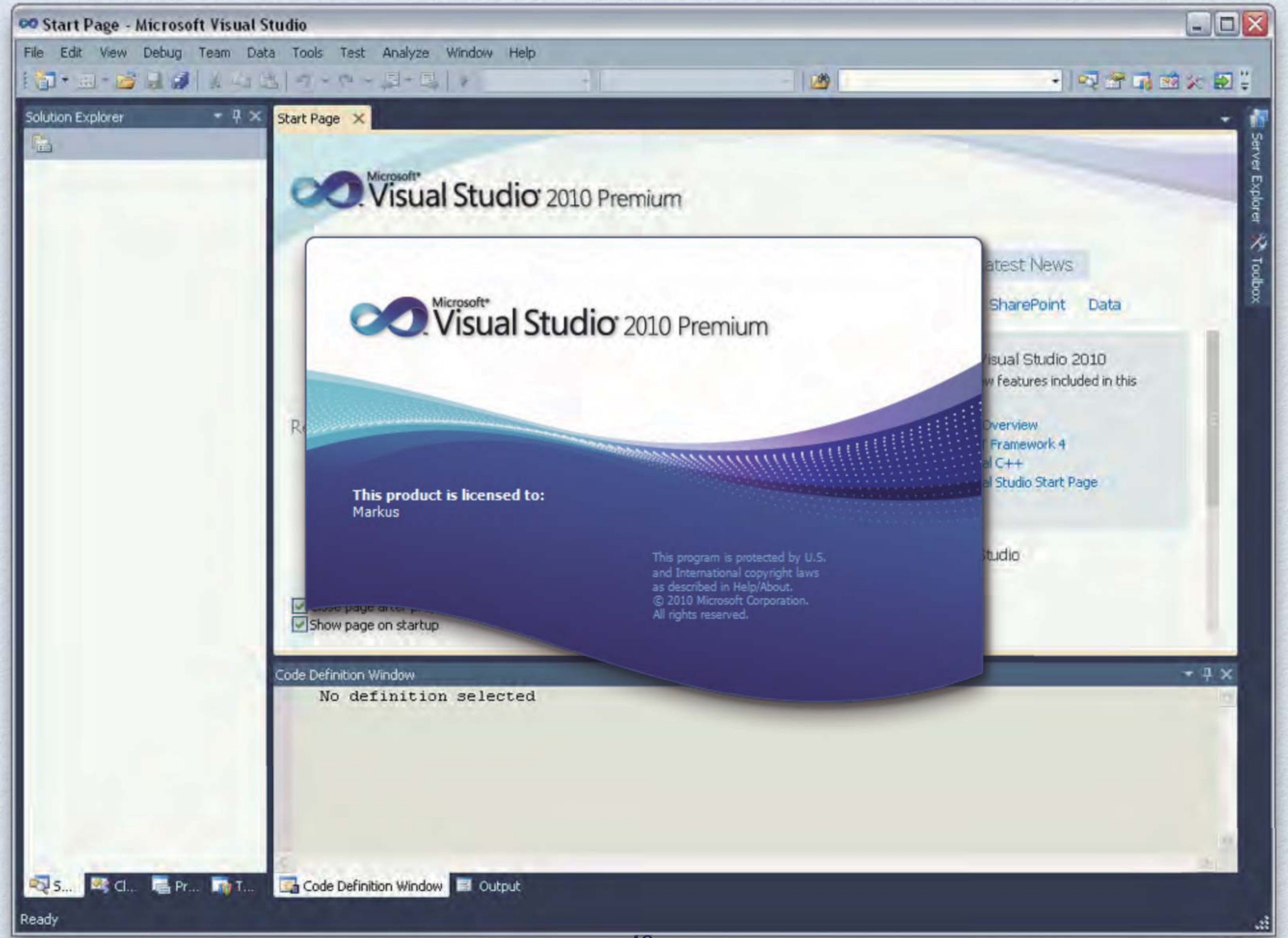
Lets make the program more responsive!



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

int main()
{
    cout << "Enter two numbers:" << endl;
    int v1, v2;
    cin >> v1 >> v2;
    cout << "The sum of " << v1 << " and " << v2
         << " is " << v1 + v2 << endl;
    return 0;
}
```

```
Sela:HIT3303 Markus$ g++ -o SimpleIO SimpleIO.cpp
Sela:HIT3303 Markus$ ./SimpleIO
Enter two numbers:
7 5
The sum of 7 and 5 is 12
Sela:HIT3303 Markus$
```

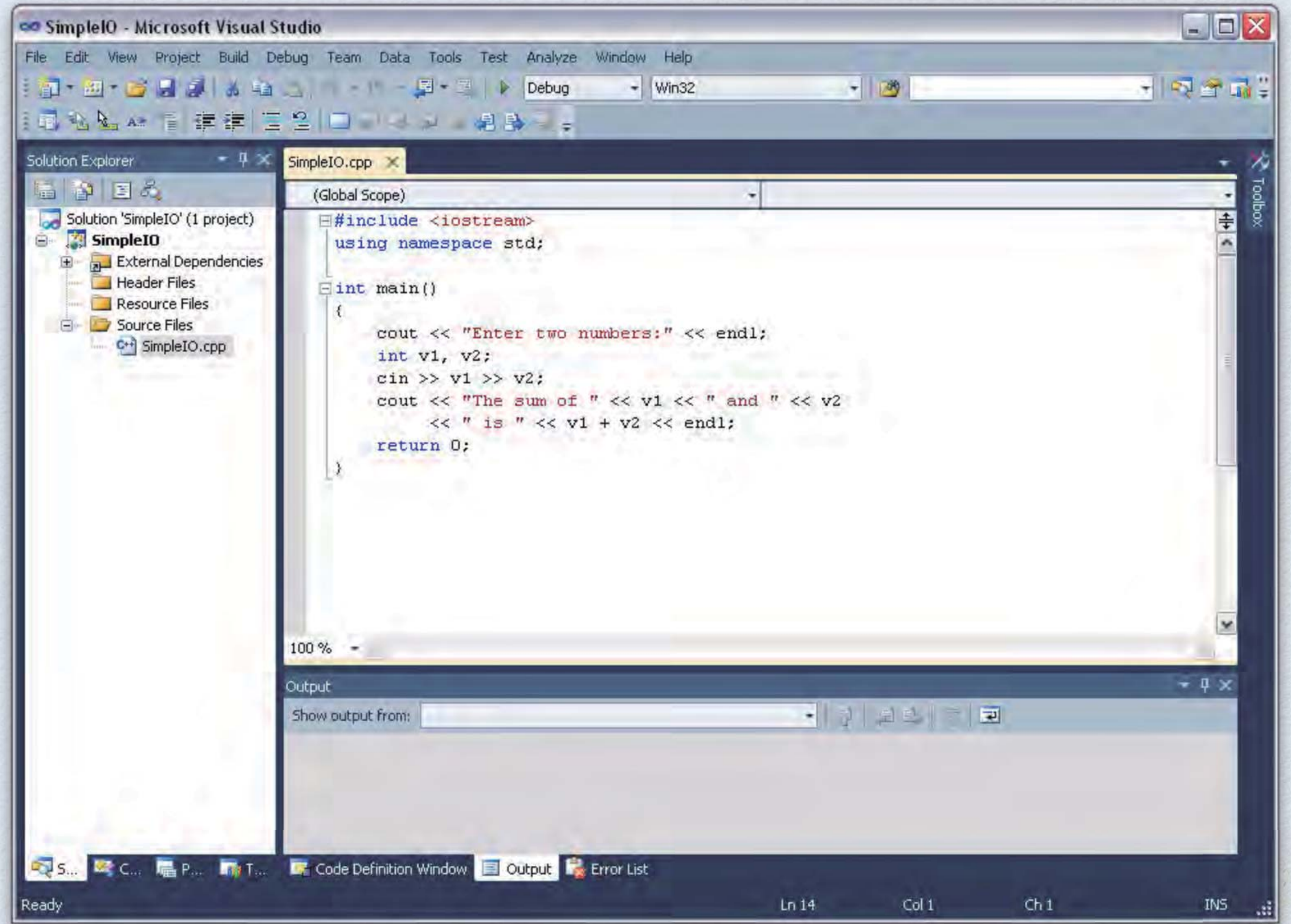
- C++ does not directly define any I/O primitives.
- I/O operations are provided by standard libraries.


Start Page - Microsoft Visual Studio
File Edit View Debug Team Data Tools Test Analyze Window Help
Solution Explorer
Start Page
Microsoft Visual Studio 2010 Premium
Latest News
SharePoint Data
Close page after project load
Show page on startup
Code Definition Window
Code Definition Window Output
Server Explorer Toolbox
Ready

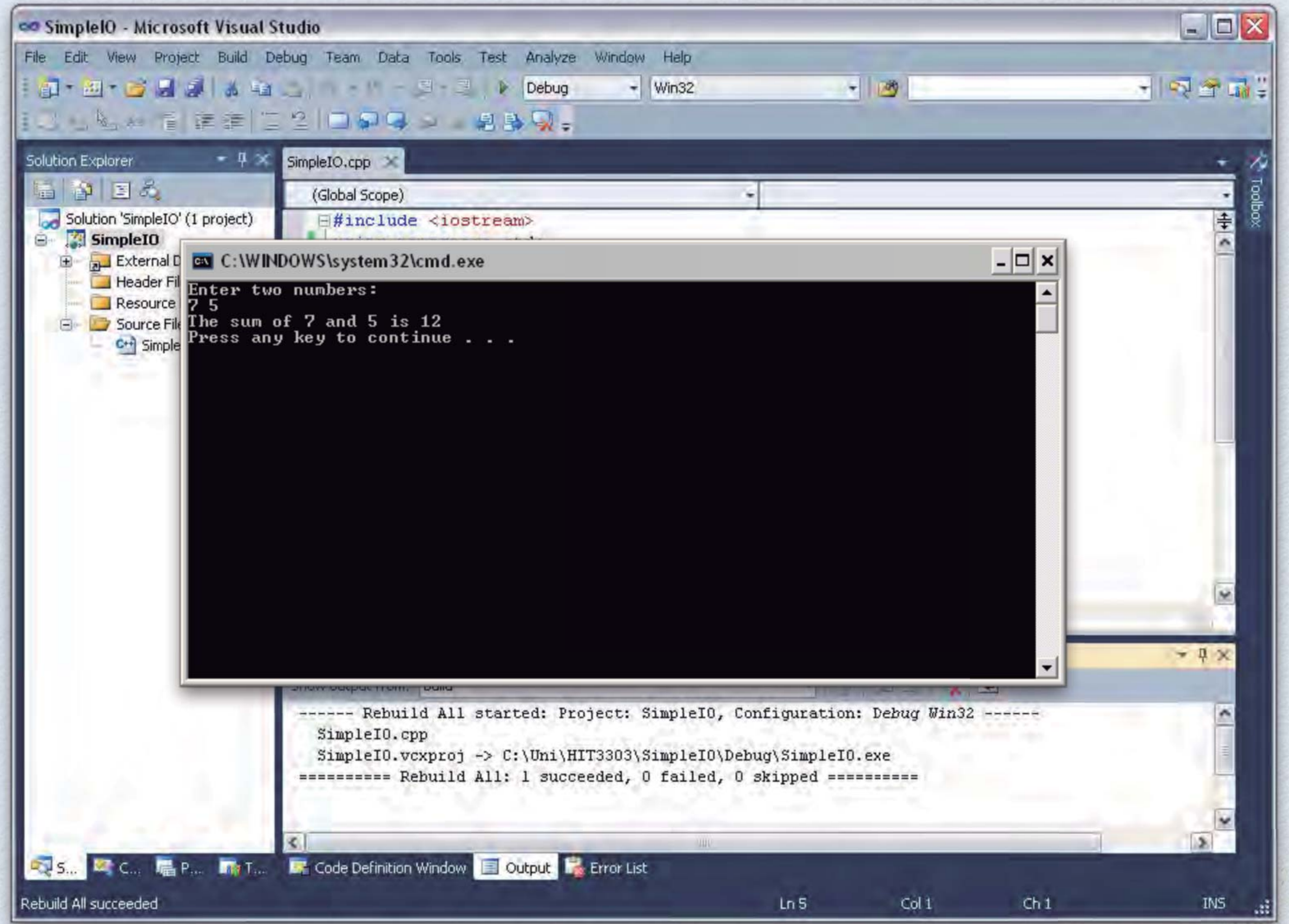
This product is licensed to:
Markus

This program is protected by U.S. and International copyright laws as described in Help/About.
© 2010 Microsoft Corporation.
All rights reserved.

```
No definition selected
```



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

int main()
{
    cout << "Enter two numbers:" << endl;
    int v1, v2;
    cin >> v1 >> v2;
    cout << "The sum of " << v1 << " and " << v2
         << " is " << v1 + v2 << endl;
    return 0;
}
```


SimpleIO - Microsoft Visual Studio
File Edit View Project Build Debug Team Data Tools Test Analyze Window Help
Debug
Win32
Solution Explorer
Solution 'SimpleIO' (1 project)
SimpleIO
SimpleIO.cpp
(Global Scope)
C:\WINDOWS\system32\cmd.exe
Code Definition Window
Output
Error List
Rebuild All succeeded
Ln 5
Col 1
Ch 1
INS

```
#include <iostream>
```

```
Enter two numbers:
7 5
The sum of 7 and 5 is 12
Press any key to continue . . .
```

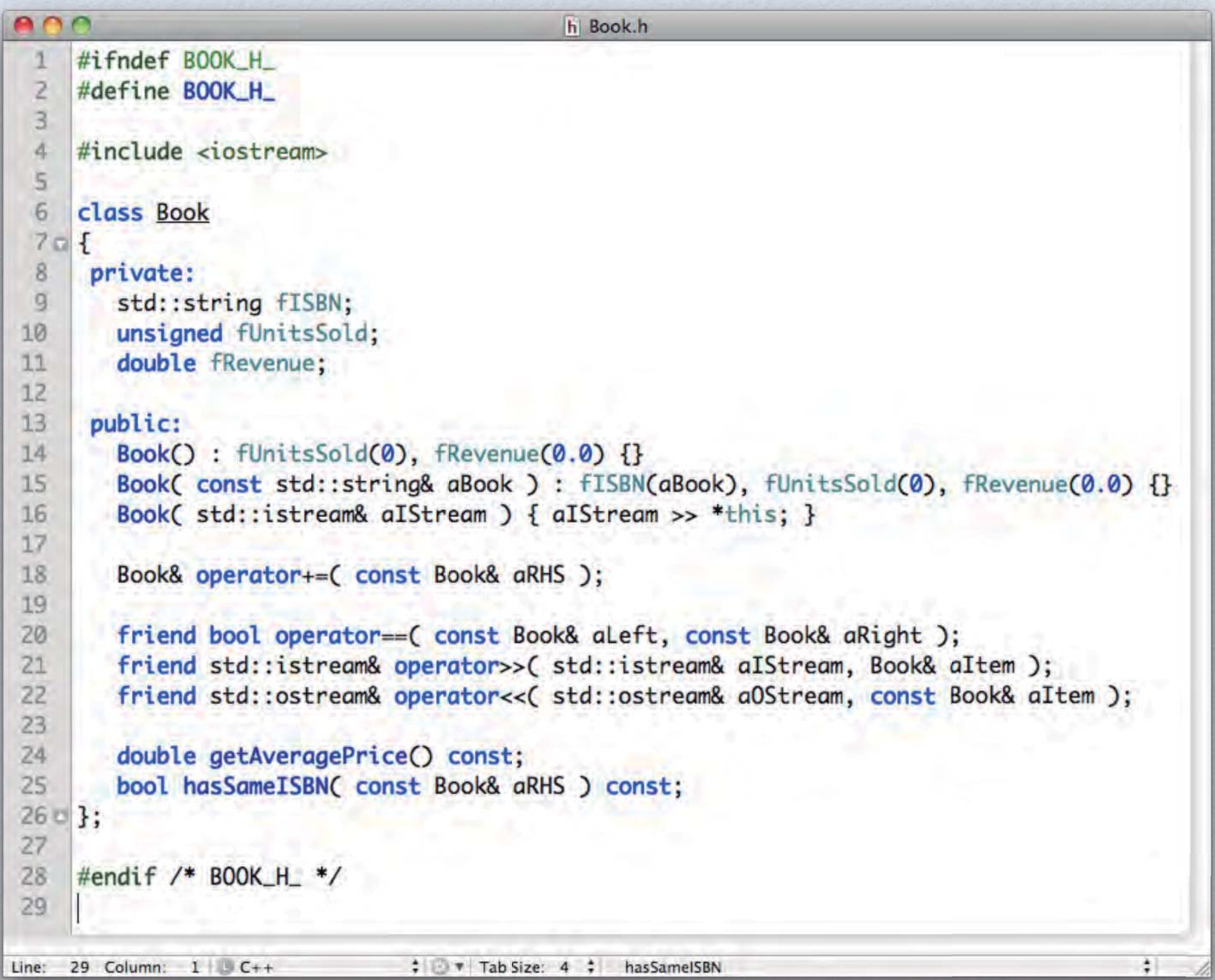
```
------ Rebuild All started: Project: SimpleIO, Configuration: Debug Win32 ------
  SimpleIO.cpp
  SimpleIO.vcxproj -> C:\Uni\HIT3303\SimpleIO\Debug\SimpleIO.exe
========== Rebuild All: 1 succeeded, 0 failed, 0 skipped ==========
```

BookStore

First Steps

- Before we can write a program in a new language, we need to know some of its basic features. C++ is no exception.
- The book store program requires us to
  - define variables
  - perform input and output
  - define a data structure (aka a class) to hold the data we are managing
  - test whether two items (aka objects) denote the same value
  - write control code to process data

C++ Class Specification



```
#ifndef BOOK_H_
#define BOOK_H_

#include <iostream>

class Book
{
 private:
   std::string fISBN;
   unsigned fUnitsSold;
   double fRevenue;

 public:
   Book() : fUnitsSold(0), fRevenue(0.0) {}
   Book( const std::string& aBook ) : fISBN(aBook), fUnitsSold(0), fRevenue(0.0) {}
   Book( std::istream& aIStream ) { aIStream >> *this; }

   Book& operator+=( const Book& aRHS );

   friend bool operator==( const Book& aLeft, const Book& aRight );
   friend std::istream& operator>>( std::istream& aIStream, Book& aItem );
   friend std::ostream& operator<<( std::ostream& aOStream, const Book& aItem );

   double getAveragePrice() const;
   bool hasSameISBN( const Book& aRHS ) const;
};

#endif /* BOOK_H_ */
```

The Structure of a Class

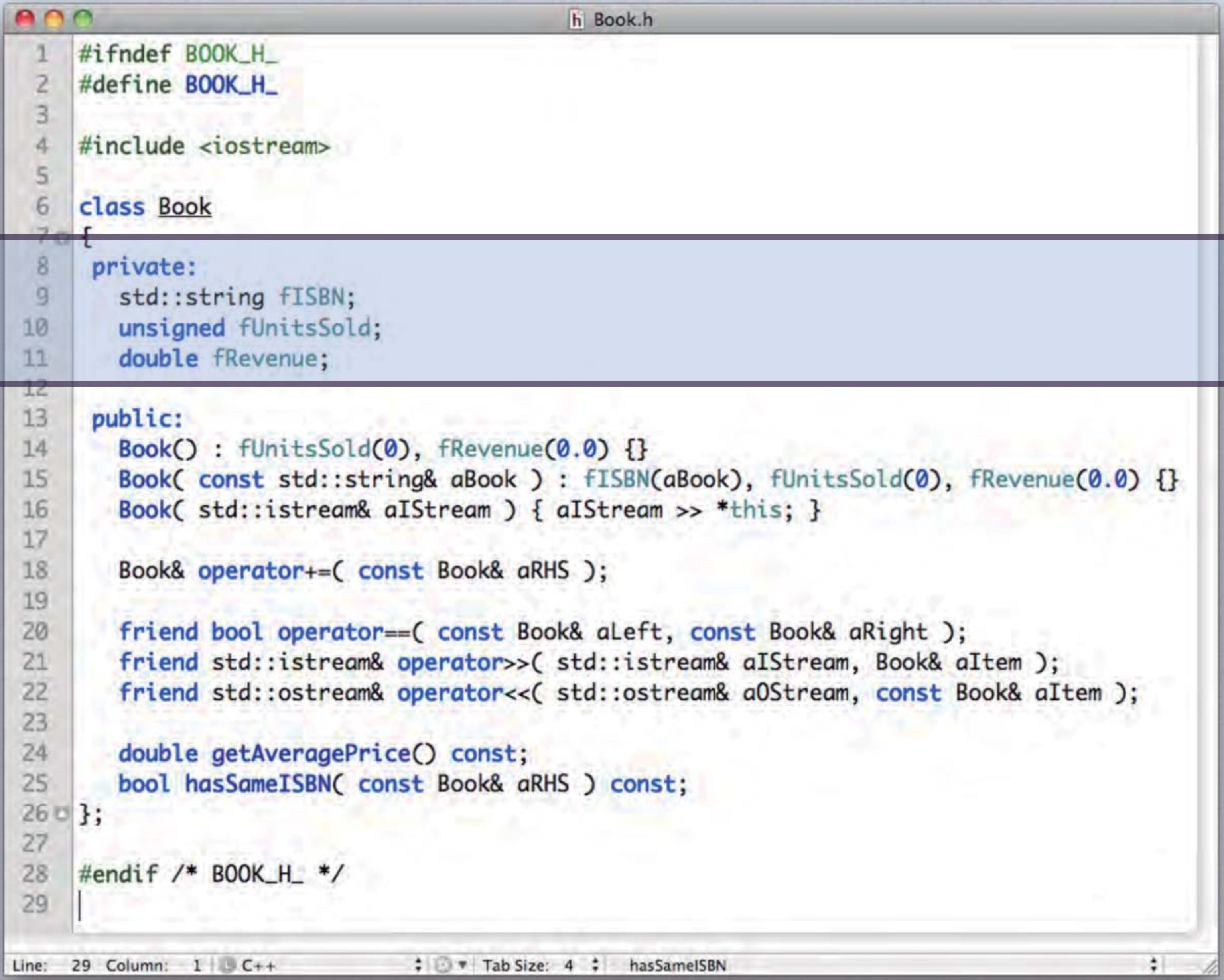
```
class X
{
private:
        // private members
protected:
        // protected members
public:
        // public members
};
```

Never forget the semicolon!

Access Modifiers

- public:
  - Public members can be accessed anywhere, including outside of the class itself.
- protected:
  - Protected members can be accessed within the class in which they are declared and within derived classes.
- private:
  - Private members can be accessed only within the class in which they are declared.

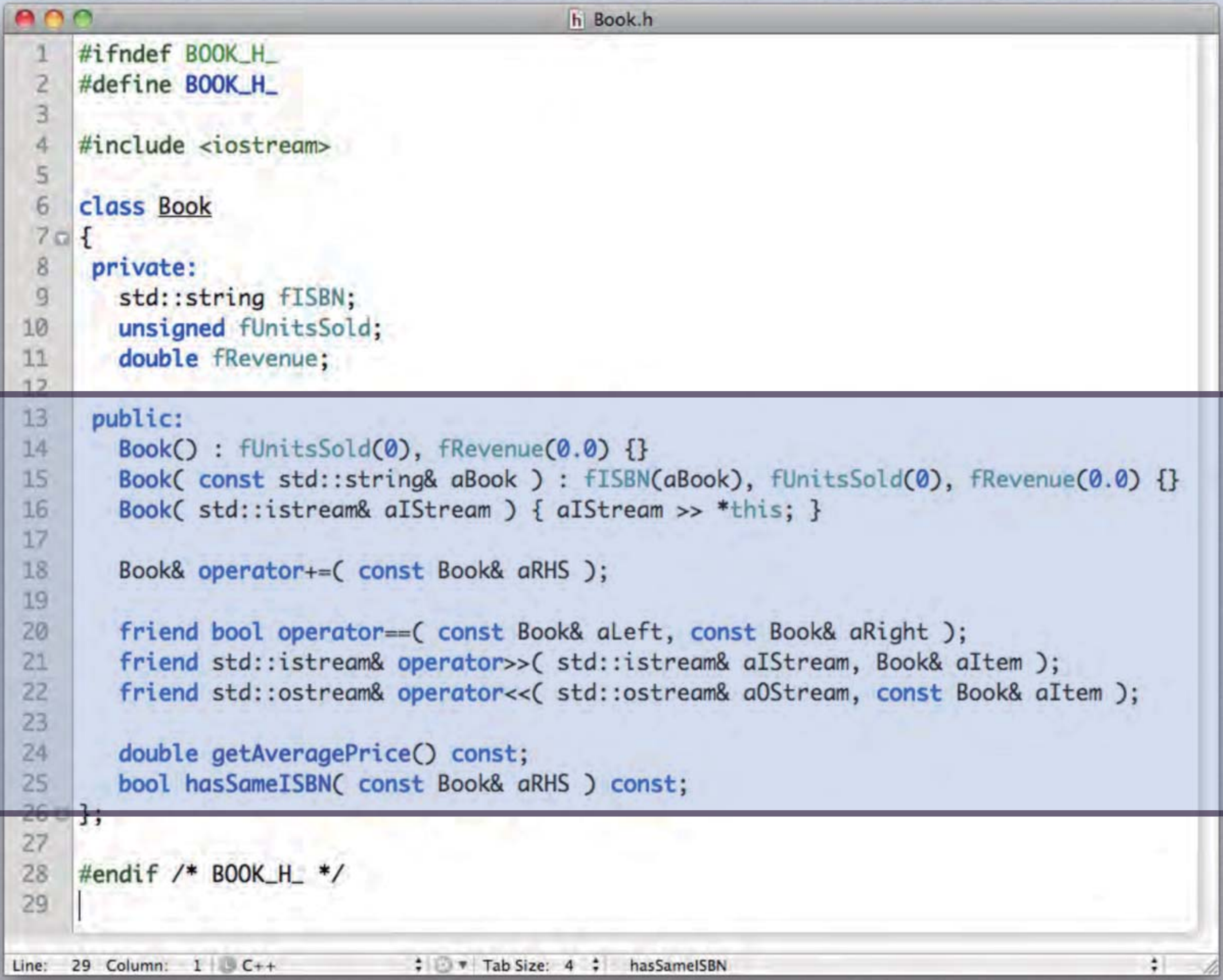
Class Book - Private Members



Book.h

```cpp
#ifndef BOOK_H_
#define BOOK_H_

#include <iostream>

class Book
{
 private:
   std::string fISBN;
   unsigned fUnitsSold;
   double fRevenue;

 public:
   Book() : fUnitsSold(0), fRevenue(0.0) {}
   Book( const std::string& aBook ) : fISBN(aBook), fUnitsSold(0), fRevenue(0.0) {}
   Book( std::istream& aIStream ) { aIStream >> *this; }

   Book& operator+=( const Book& aRHS );

   friend bool operator==( const Book& aLeft, const Book& aRight );
   friend std::istream& operator>>( std::istream& aIStream, Book& aItem );
   friend std::ostream& operator<<( std::ostream& aOStream, const Book& aItem );

   double getAveragePrice() const;
   bool hasSameISBN( const Book& aRHS ) const;
};

#endif /* BOOK_H_ */
```

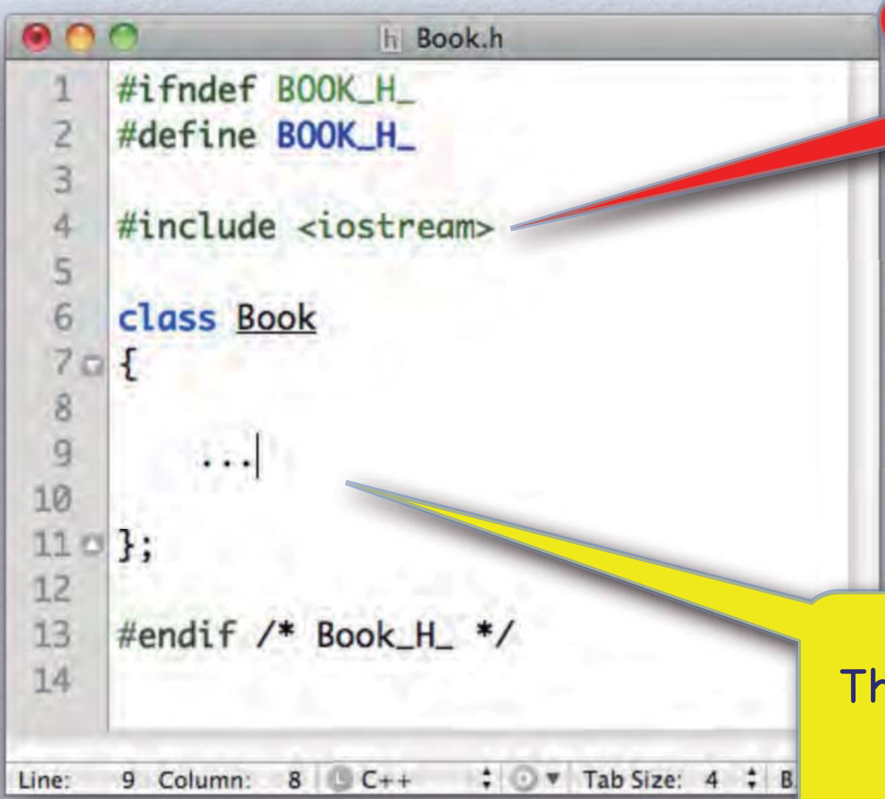
Class Book - Public Members



Book.h

```
#ifndef BOOK_H_
#define BOOK_H_

#include <iostream>

class Book
{
 private:
   std::string fISBN;
   unsigned fUnitsSold;
   double fRevenue;

 public:
   Book() : fUnitsSold(0), fRevenue(0.0) {}
   Book( const std::string& aBook ) : fISBN(aBook), fUnitsSold(0), fRevenue(0.0) {}
   Book( std::istream& aIStream ) { aIStream >> *this; }

   Book& operator+=( const Book& aRHS );

   friend bool operator==( const Book& aLeft, const Book& aRight );
   friend std::istream& operator>>( std::istream& aIStream, Book& aItem );
   friend std::ostream& operator<<( std::ostream& aOStream, const Book& aItem );

   double getAveragePrice() const;
   bool hasSameISBN( const Book& aRHS ) const;
};

#endif /* BOOK_H_ */

```

Include File: Book.h

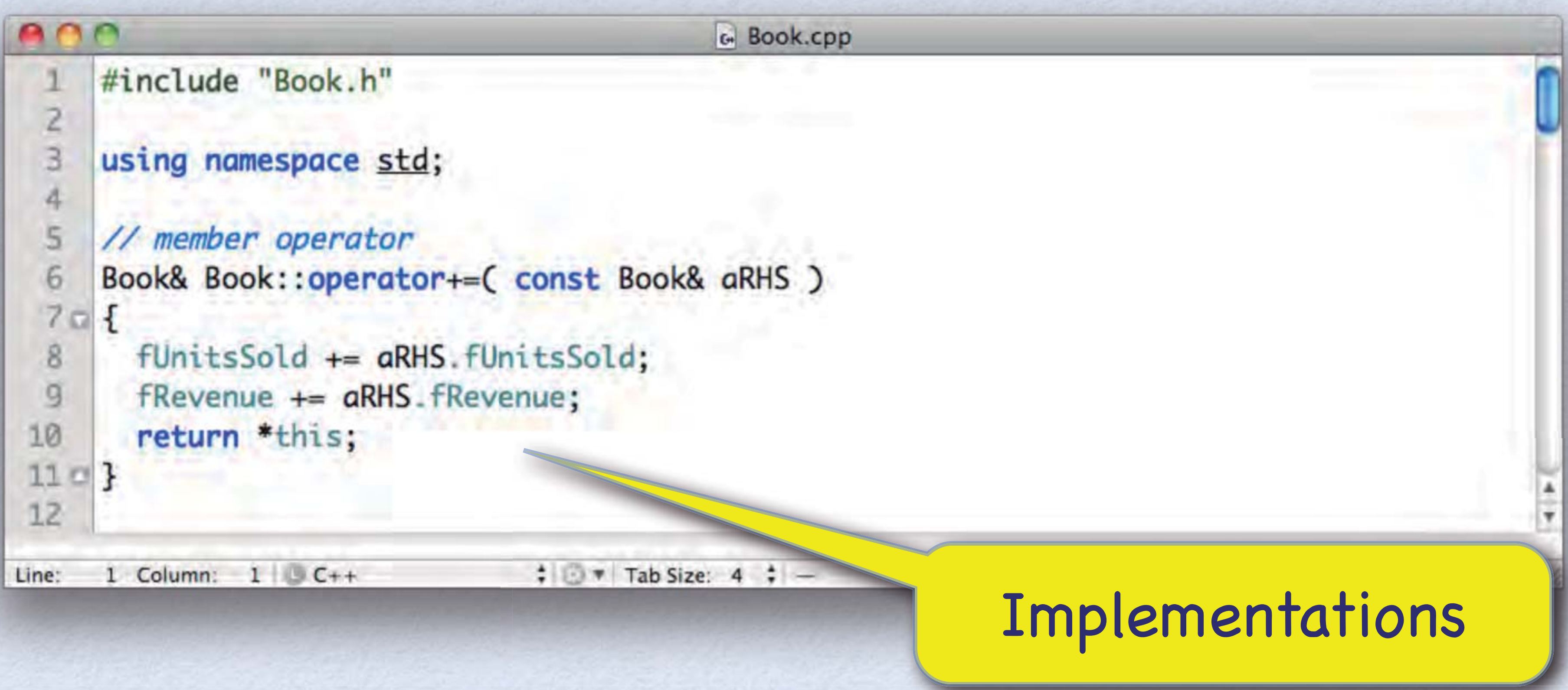


```
#ifndef BOOK_H_
#define BOOK_H_

#include <iostream>

class Book
{

    ...

};

#endif /* Book_H_ */
```

We do not select any namespace yet!

The implementation goes to Book.cpp

Implementation File: Book.cpp



```
#include "Book.h"

using namespace std;

// member operator
Book& Book::operator+=( const Book& aRHS )
{
  fUnitsSold += aRHS.fUnitsSold;
  fRevenue += aRHS.fRevenue;
  return *this;
}

```

C++ Code Organization

- Classes are defined in include files (i.e., .h).
- Class members are implemented in source files (i.e., .cpp).
- There are exceptions (as usual), when working with templates.

Standard Boilerplate Code

Guard against repeated inclusion

```
#ifndef HEADER_H_
#define HEADER_H_


   /* Body of Header */


#endif /* HEADER_H_ */
```

#pragma once (Visual Studio)

```
#pragma once


  /* Body of Header */
```

Guard against repeated inclusion

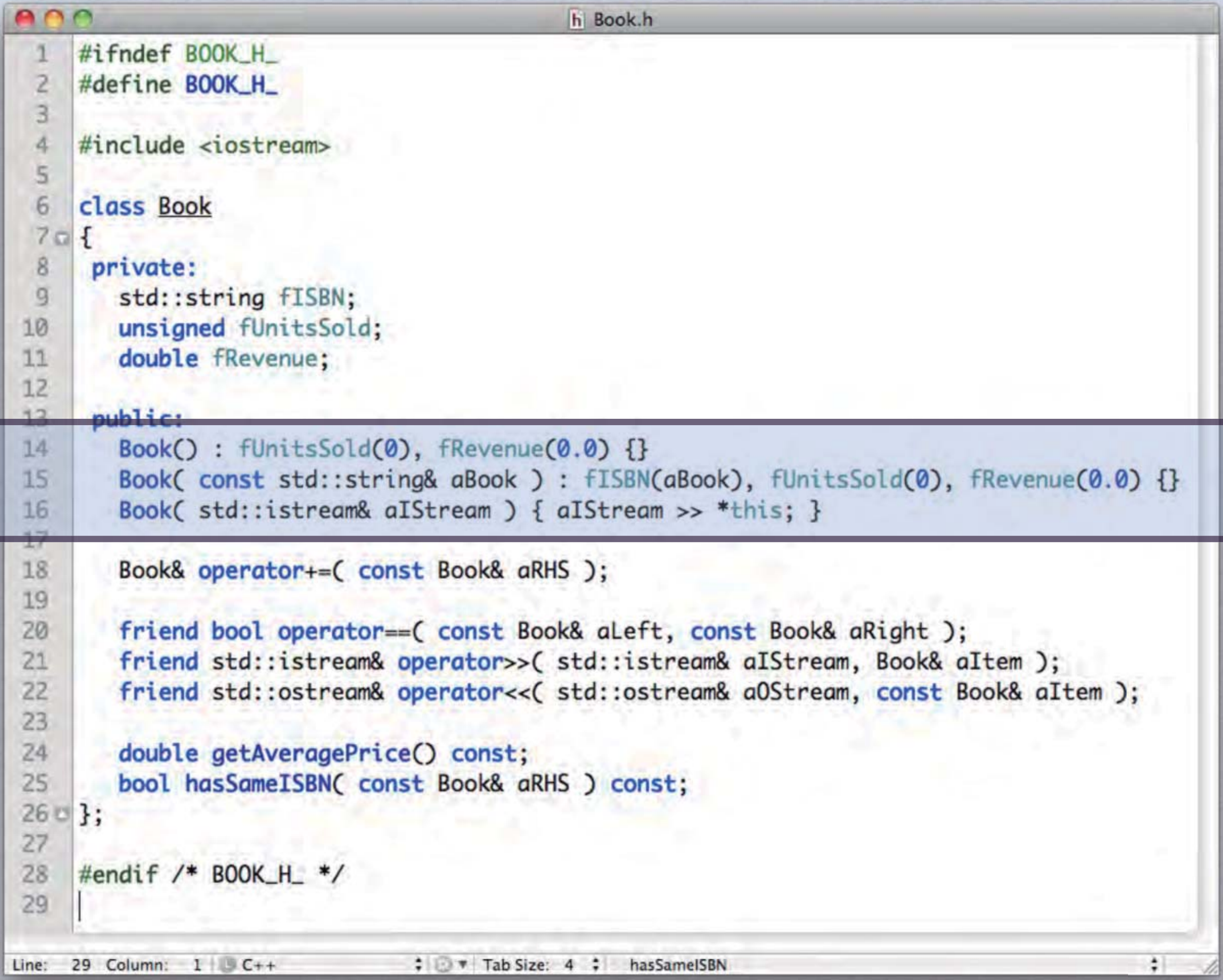
Constructors

- Constructors may be overloaded.
- The concrete constructor arguments determine which constructor to use.
- Constructors are executed automatically whenever a new object is created.

Constructor Initializer

- A constructor initializer is a comma-separated list of member initializers, which is declared between the signature of the constructor and its body.

Class Book – Constructors



Book.h

```
#ifndef BOOK_H_
#define BOOK_H_

#include <iostream>

class Book
{
 private:
   std::string fISBN;
   unsigned fUnitsSold;
   double fRevenue;

 public:
   Book() : fUnitsSold(0), fRevenue(0.0) {}
   Book( const std::string& aBook ) : fISBN(aBook), fUnitsSold(0), fRevenue(0.0) {}
   Book( std::istream& aIStream ) { aIStream >> *this; }

   Book& operator+=( const Book& aRHS );

   friend bool operator==( const Book& aLeft, const Book& aRight );
   friend std::istream& operator>>( std::istream& aIStream, Book& aItem );
   friend std::ostream& operator<<( std::ostream& aOStream, const Book& aItem );

   double getAveragePrice() const;
   bool hasSameISBN( const Book& aRHS ) const;
};

#endif /* BOOK_H_ */
```

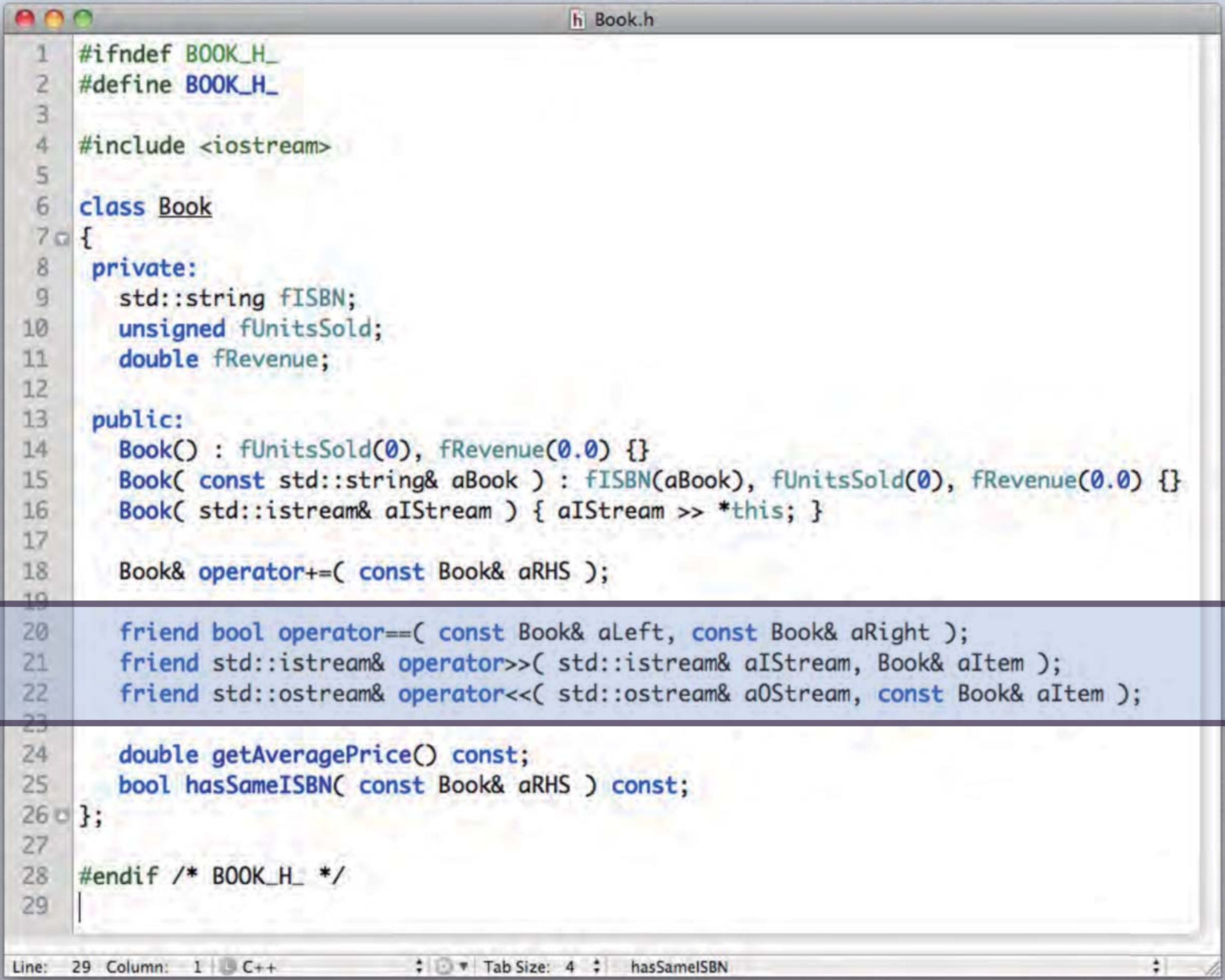
Friends

- Friends are allowed to access private members of classes.
- A class declares its friends explicitly.
- Friends enable uncontrolled access to members.
- The friend mechanism induces a particular programming (C++) style.
- The friend mechanism is not object-oriented!
- I/O depends on the friend mechanism.

The Friend Mechanism

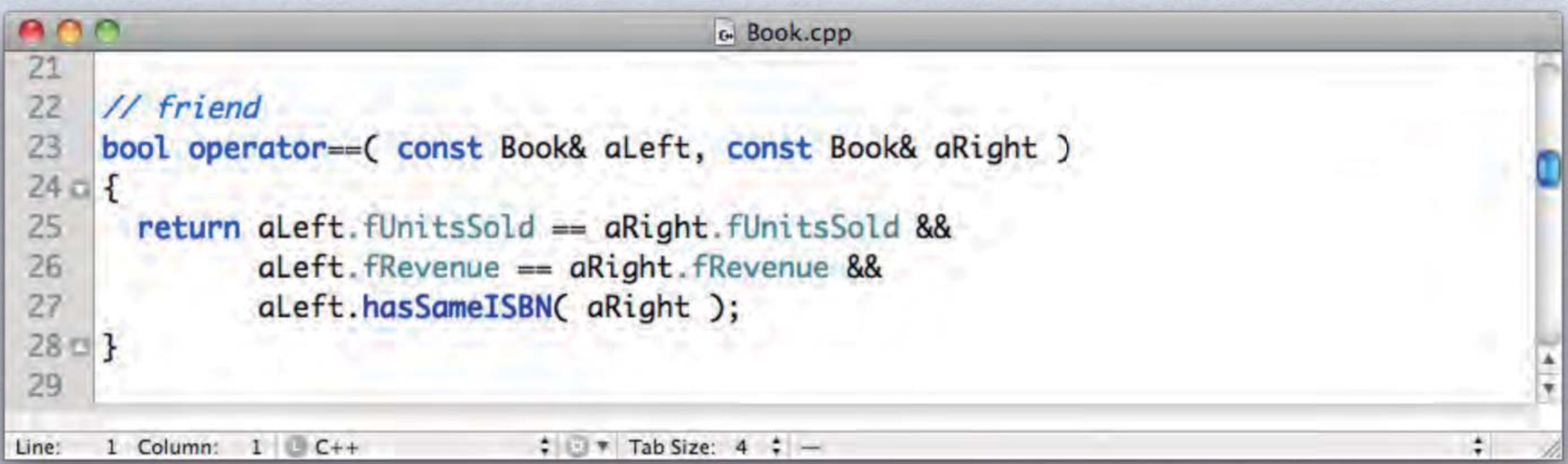
Friends are self-contained procedures (or functions) that do not belong to a specific class, but have access to the members of a class, when this class declares those procedures as friends.

Class Book - The Friends



```
#ifndef BOOK_H_
#define BOOK_H_

#include <iostream>

class Book
{
 private:
   std::string fISBN;
   unsigned fUnitsSold;
   double fRevenue;

 public:
   Book() : fUnitsSold(0), fRevenue(0.0) {}
   Book( const std::string& aBook ) : fISBN(aBook), fUnitsSold(0), fRevenue(0.0) {}
   Book( std::istream& aIStream ) { aIStream >> *this; }

   Book& operator+=( const Book& aRHS );

   friend bool operator==( const Book& aLeft, const Book& aRight );
   friend std::istream& operator>>( std::istream& aIStream, Book& aItem );
   friend std::ostream& operator<<( std::ostream& aOStream, const Book& aItem );

   double getAveragePrice() const;
   bool hasSameISBN( const Book& aRHS ) const;
};

#endif /* BOOK_H_ */

```

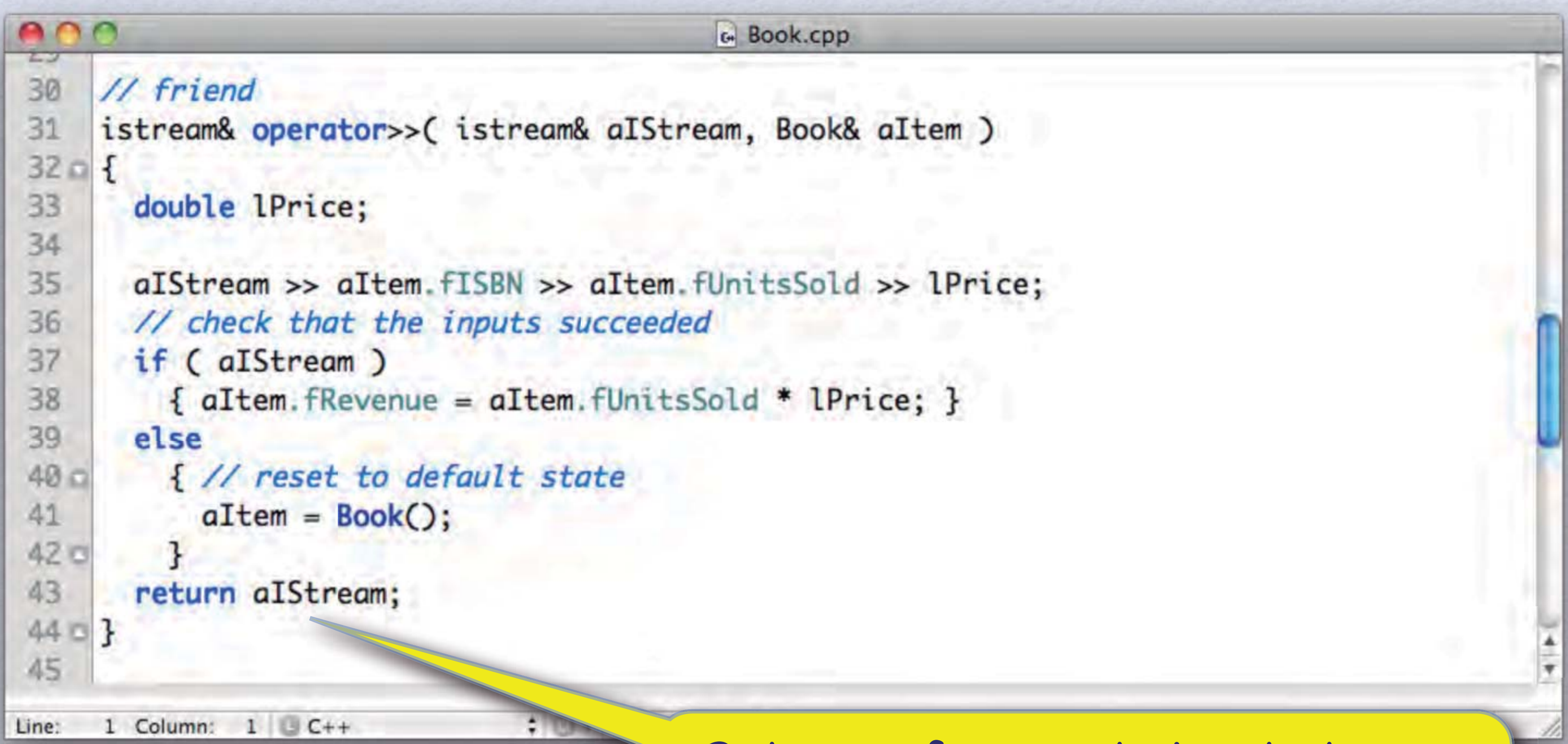
The Equivalence Operator ==



```
// friend
bool operator==( const Book& aLeft, const Book& aRight )
{
  return aLeft.fUnitsSold == aRight.fUnitsSold &&
         aLeft.fRevenue == aRight.fRevenue &&
         aLeft.hasSameISBN( aRight );
}
```

- The Boolean operator == defines a structural equivalence test for Book objects.
- We use const references for the Book arguments to pass Book objects by reference rather than copying their values into the stack frame of the operator ==.

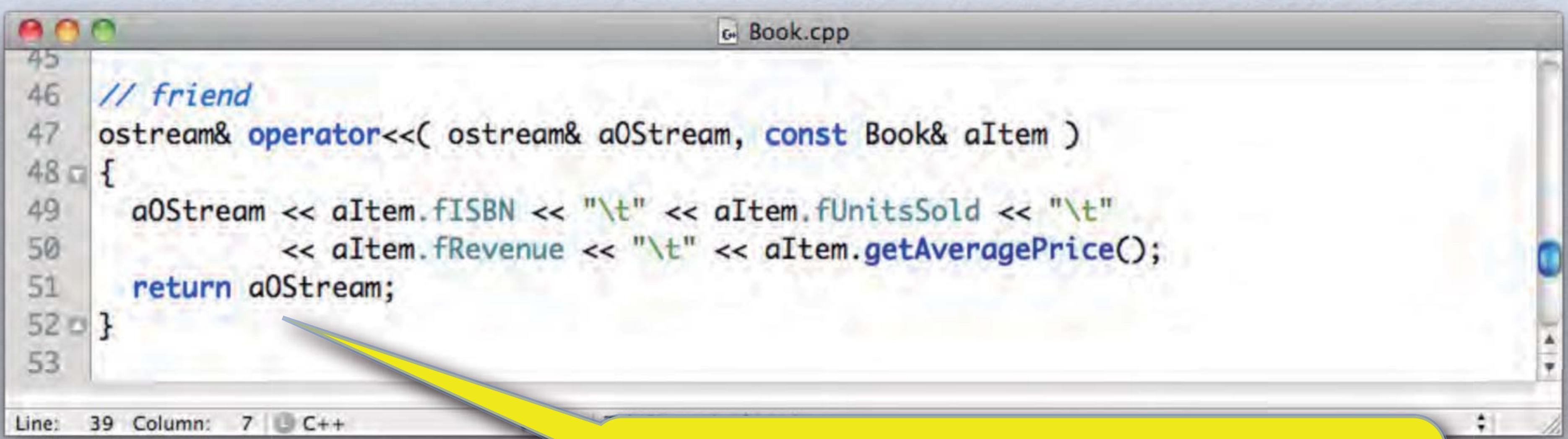
The Input Operator >>


Book.cpp

```
// friend
istream& operator>>( istream& aIStream, Book& aItem )
{
  double lPrice;

  aIStream >> aItem.fISBN >> aItem.fUnitsSold >> lPrice;
  // check that the inputs succeeded
  if ( aIStream )
    { aItem.fRevenue = aItem.fUnitsSold * lPrice; }
  else
    { // reset to default state
      aItem = Book();
    }
  return aIStream;
}
```

Return reference to input stream.

The Output Operator <<

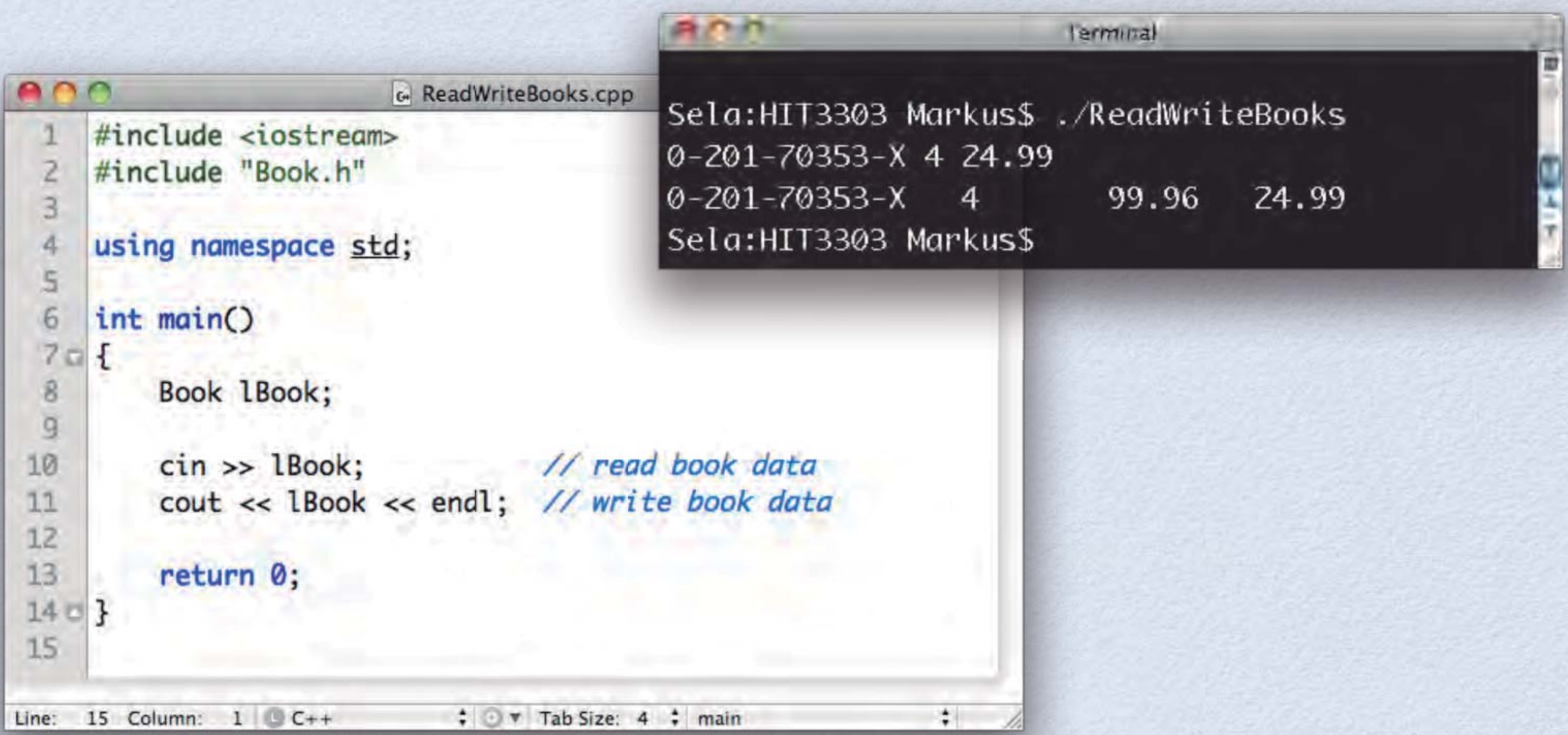


Book.cpp

```
// friend
ostream& operator<<( ostream& aOStream, const Book& aItem )
{
  aOStream << aItem.fISBN << "\t" << aItem.fUnitsSold << "\t"
           << aItem.fRevenue << "\t" << aItem.getAveragePrice();
  return aOStream;
}
```

Return reference to output stream.

ReadWriteBooks



ReadWriteBooks.cpp

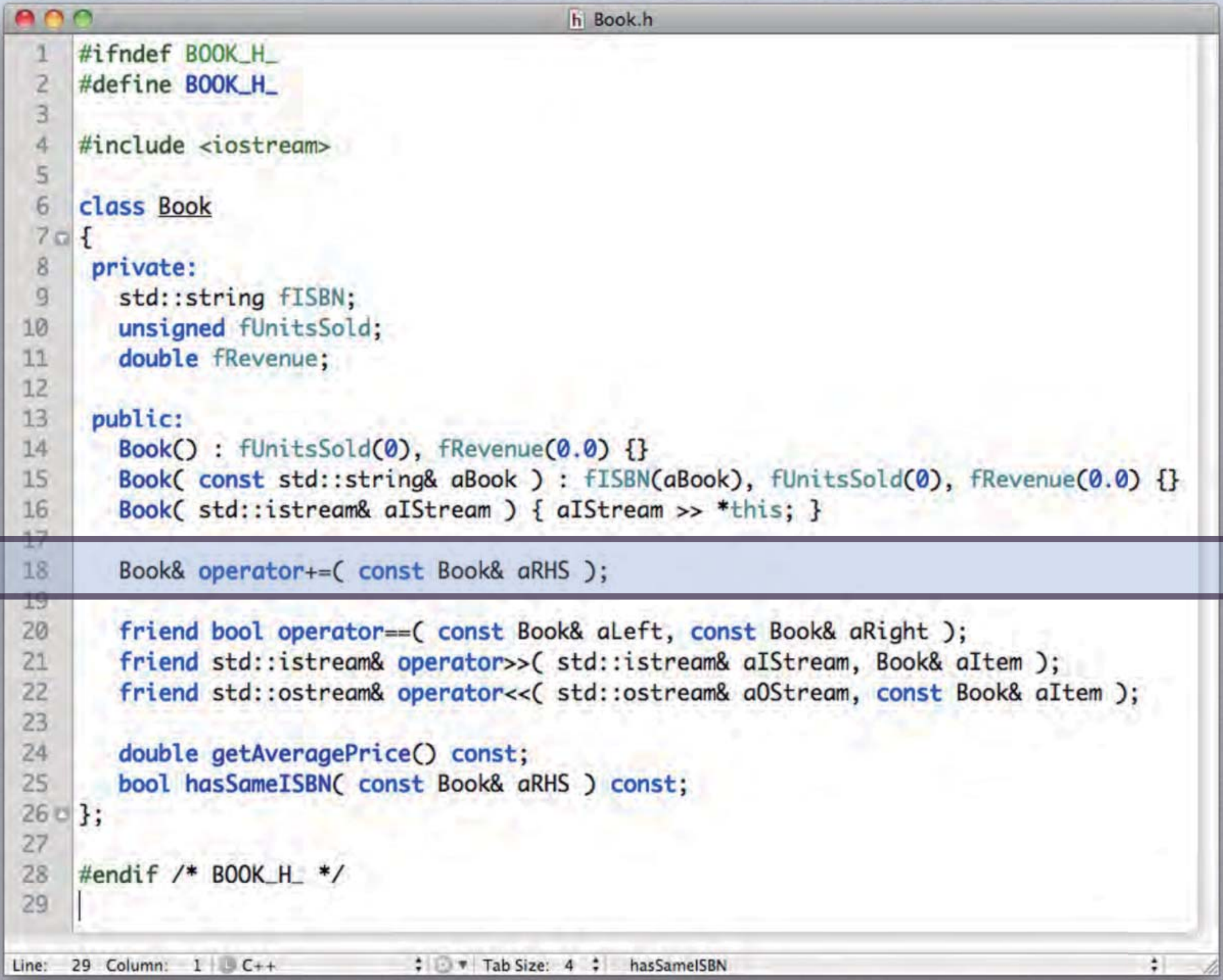
```cpp
#include <iostream>
#include "Book.h"

using namespace std;

int main()
{
    Book lBook;

    cin >> lBook;             // read book data
    cout << lBook << endl;    // write book data

    return 0;
}

```

Line: 15 Column: 1 C++ Tab Size: 4 main

Terminal

```
Sela:HIT3303 Markus$ ./ReadWriteBooks
0-201-70353-X 4 24.99
0-201-70353-X    4        99.96    24.99
Sela:HIT3303 Markus$
```

Class Book - Member Operator



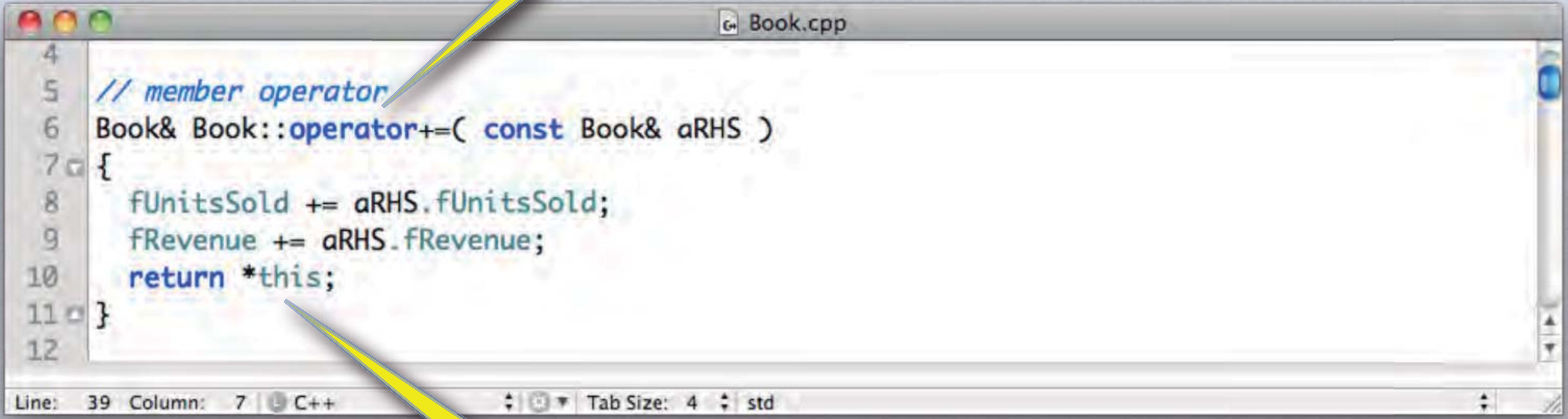
Book.h

```
#ifndef BOOK_H_
#define BOOK_H_

#include <iostream>

class Book
{
 private:
   std::string fISBN;
   unsigned fUnitsSold;
   double fRevenue;

 public:
   Book() : fUnitsSold(0), fRevenue(0.0) {}
   Book( const std::string& aBook ) : fISBN(aBook), fUnitsSold(0), fRevenue(0.0) {}
   Book( std::istream& aIStream ) { aIStream >> *this; }

   Book& operator+=( const Book& aRHS );

   friend bool operator==( const Book& aLeft, const Book& aRight );
   friend std::istream& operator>>( std::istream& aIStream, Book& aItem );
   friend std::ostream& operator<<( std::ostream& aOStream, const Book& aItem );

   double getAveragePrice() const;
   bool hasSameISBN( const Book& aRHS ) const;
};

#endif /* BOOK_H_ */

```

Line: 29 Column: 1 C++ Tab Size: 4 hasSameISBN

The Member Operator +=

The operator += is defined as a member of class Book!



Book.cpp

```cpp

// member operator
Book& Book::operator+=( const Book& aRHS )
{
  fUnitsSold += aRHS.fUnitsSold;
  fRevenue += aRHS.fRevenue;
  return *this;
}

```

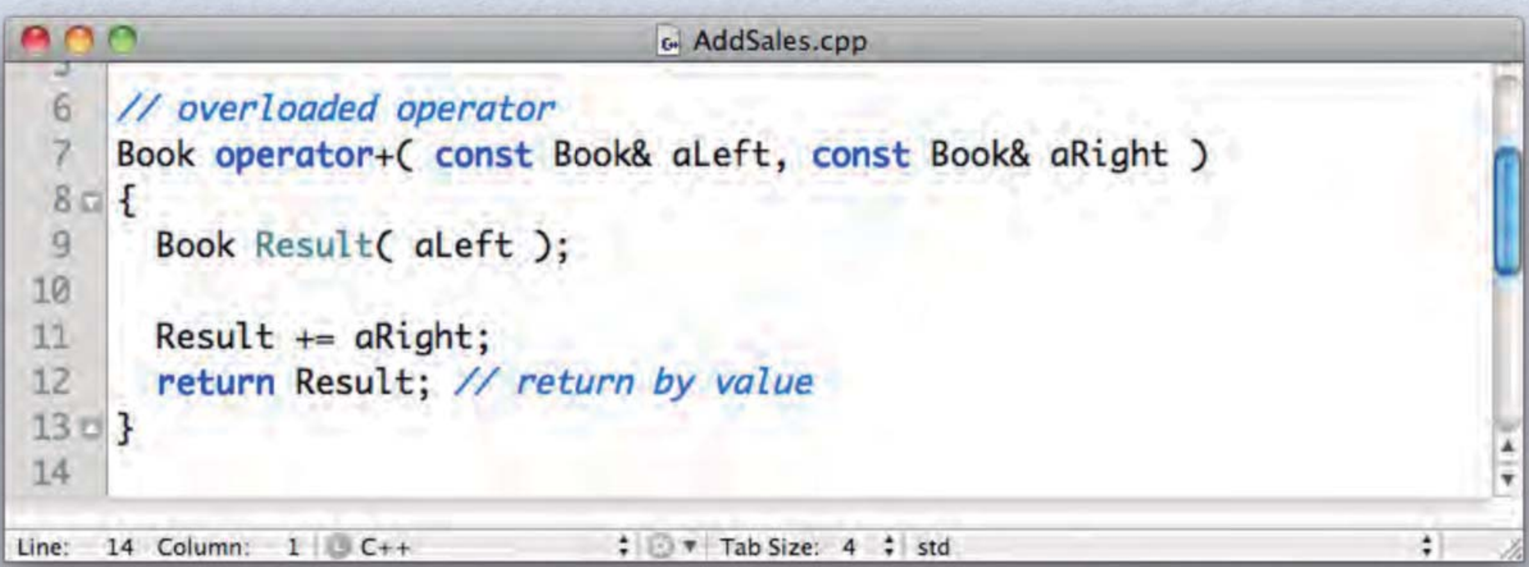
Line: 39 Column: 7 C++ Tab Size: 4 std

Return reference to receiver.

Operator Overloading

- C++ supports operator overloading.
- Overloaded operators are like normal functions, but are defined using a pre-defined operator symbol.
- You cannot change the priority and associativity of an operator.
- Operators are selected by the compiler based on the static types of the specified operands.

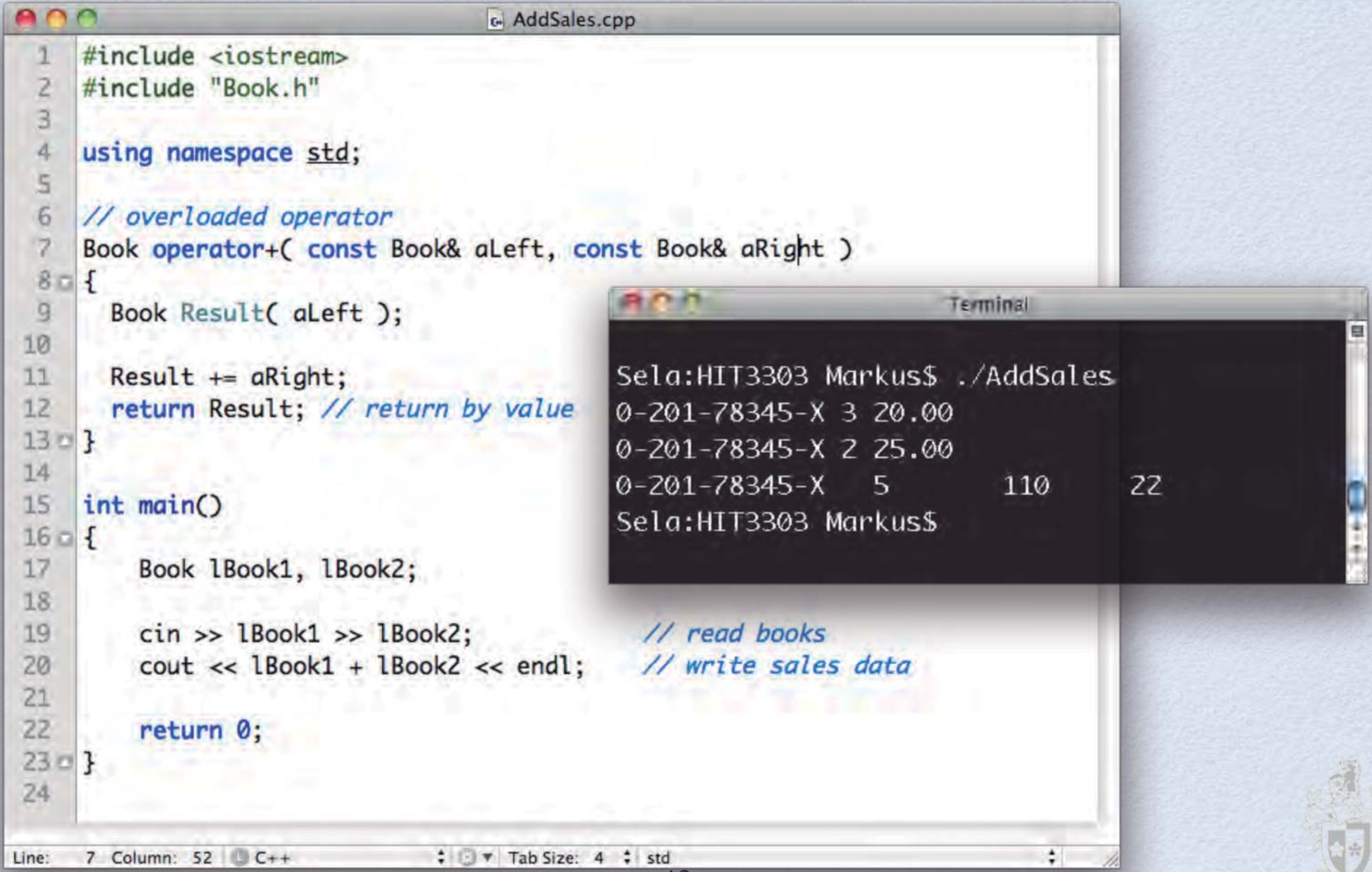
The Overloaded Operator +



```
// overloaded operator
Book operator+( const Book& aLeft, const Book& aRight )
{
  Book Result( aLeft );

  Result += aRight;
  return Result; // return by value
}

```

operator+ : (Book,Book) $\rightarrow$ Book

AddSales



AddSales.cpp

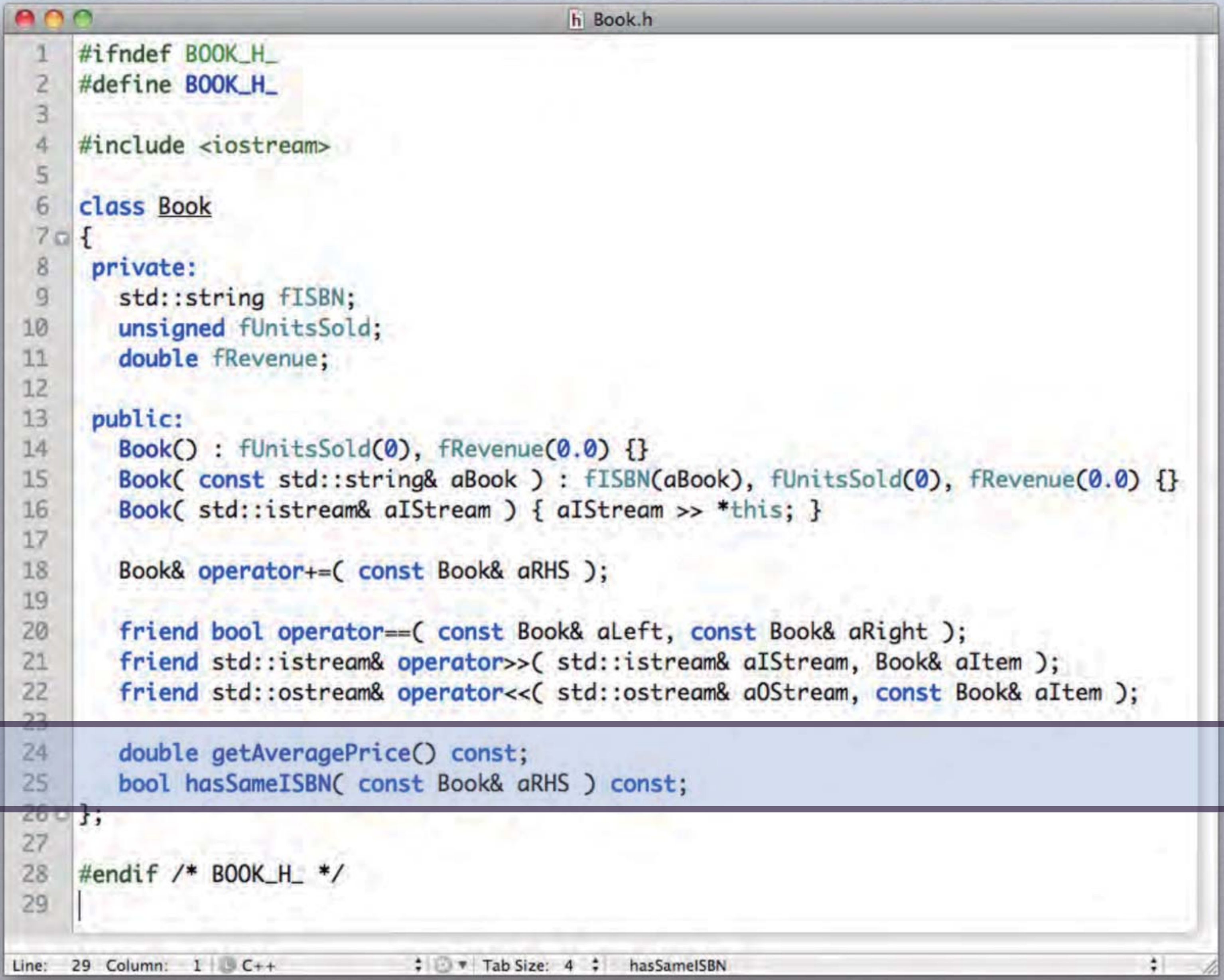
```cpp
#include <iostream>
#include "Book.h"

using namespace std;

// overloaded operator
Book operator+( const Book& aLeft, const Book& aRight )
{
  Book Result( aLeft );

  Result += aRight;
  return Result; // return by value
}

int main()
{
    Book lBook1, lBook2;

    cin >> lBook1 >> lBook2;              // read books
    cout << lBook1 + lBook2 << endl;      // write sales data

    return 0;
}
```

Line: 7 Column: 52 C++ Tab Size: 4 std

Terminal

```
Sela:HIT3303 Markus$ ./AddSales
0-201-78345-X 3 20.00
0-201-78345-X 2 25.00
0-201-78345-X   5       110     22
Sela:HIT3303 Markus$
```

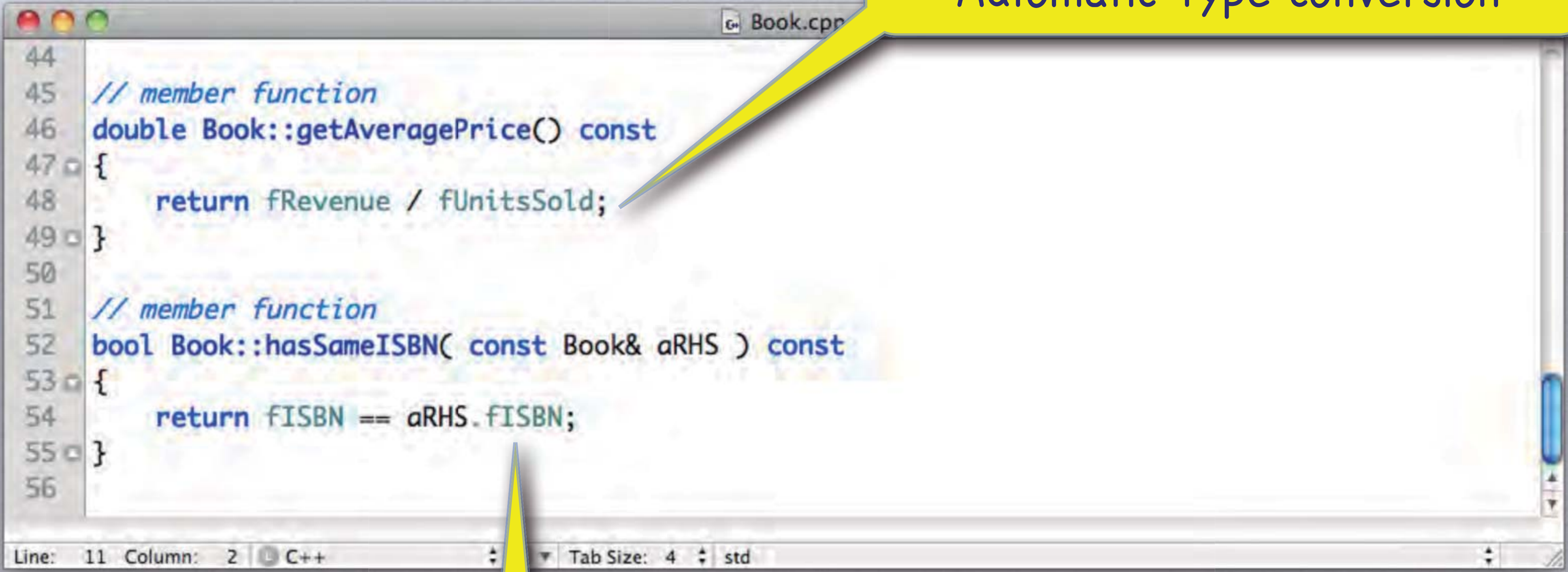
Class Book - Member Functions



Book.h

```
#ifndef BOOK_H_
#define BOOK_H_

#include <iostream>

class Book
{
 private:
   std::string fISBN;
   unsigned fUnitsSold;
   double fRevenue;

 public:
   Book() : fUnitsSold(0), fRevenue(0.0) {}
   Book( const std::string& aBook ) : fISBN(aBook), fUnitsSold(0), fRevenue(0.0) {}
   Book( std::istream& aIStream ) { aIStream >> *this; }

   Book& operator+=( const Book& aRHS );

   friend bool operator==( const Book& aLeft, const Book& aRight );
   friend std::istream& operator>>( std::istream& aIStream, Book& aItem );
   friend std::ostream& operator<<( std::ostream& aOStream, const Book& aItem );

   double getAveragePrice() const;
   bool hasSameISBN( const Book& aRHS ) const;
};

#endif /* BOOK_H_ */
```

The Member Functions



```
// member function
double Book::getAveragePrice() const
{
    return fRevenue / fUnitsSold;
}

// member function
bool Book::hasSameISBN( const Book& aRHS ) const
{
    return fISBN == aRHS.fISBN;
}
```

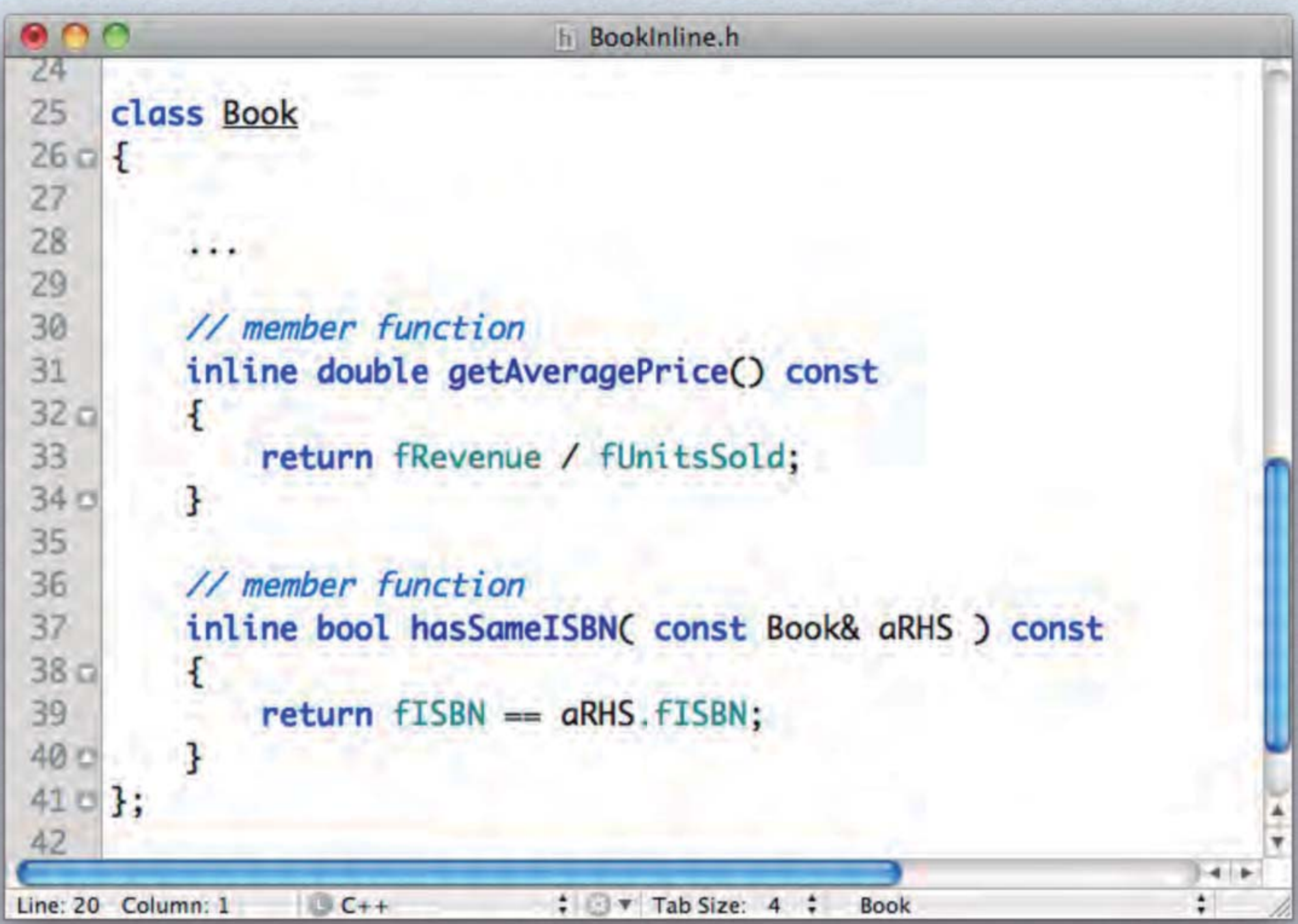
Automatic type conversion

Private member variables are visible within the scope of class Book.

A Small Note: Inlining

- C++ offers function inlining that may reduce the calling overhead associated with a function.
- Inlining increases the code size and may not be applicable (e.g., recursive or virtual functions).
- Inlining can give better cache performance, but too many inlined functions can result in a large code size and page faults, hence defeating the original aim of inlining. (Inlining is not useful for embedded systems, where large binaries are not preferred.)
- The compiler may choose to ignore inline requests.

Inlined Member Functions


BookInline.h

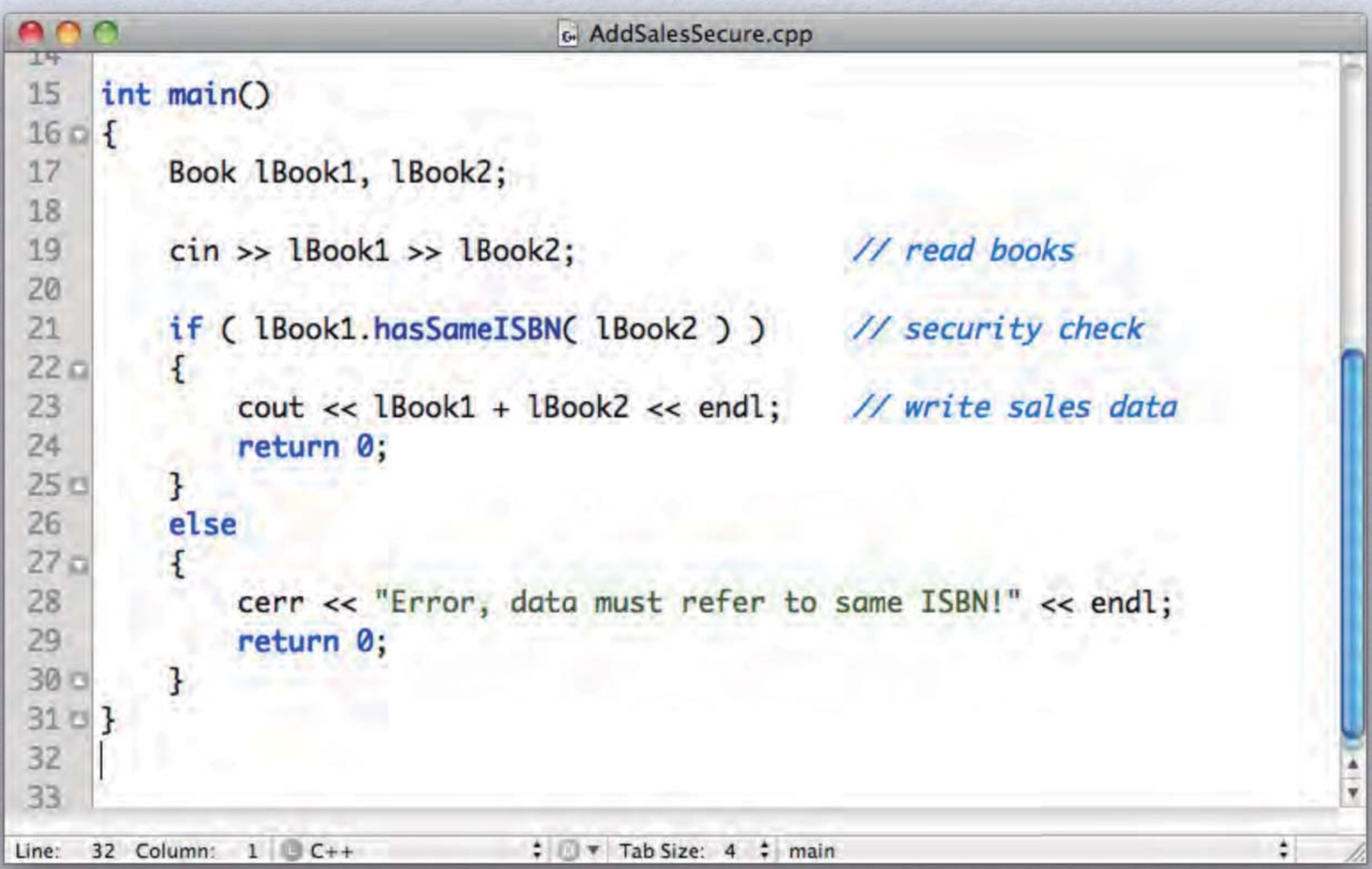
```
class Book
{

    ...

    // member function
    inline double getAveragePrice() const
    {
        return fRevenue / fUnitsSold;
    }

    // member function
    inline bool hasSameISBN( const Book& aRHS ) const
    {
        return fISBN == aRHS.fISBN;
    }
};
```

Line: 20 Column: 1 C++ Tab Size: 4 Book

- For inlining to work the functions need to be defined in the header file, that is, within the class specification.

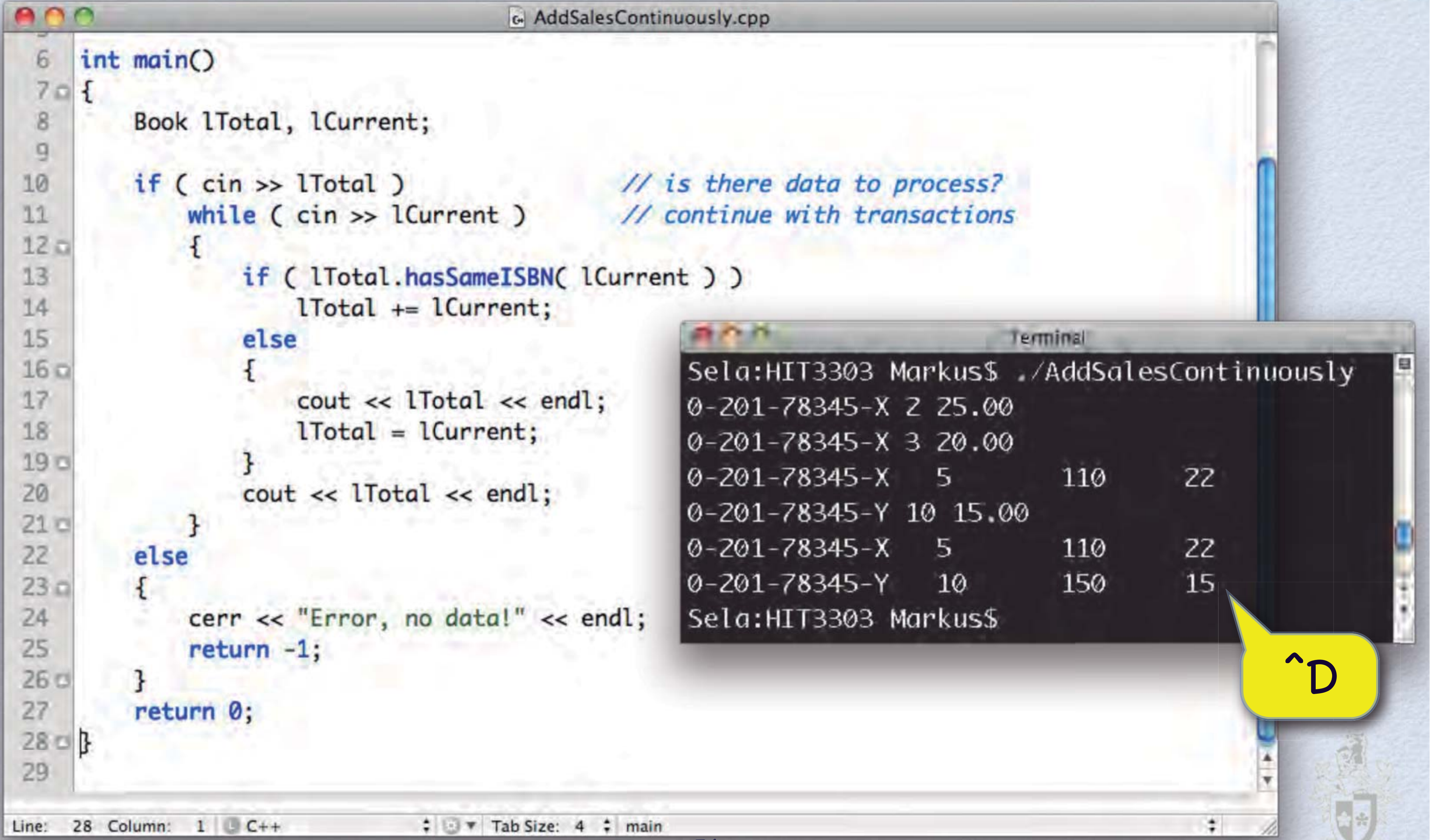
Control Features At Work

AddSalesSecure



```
int main()
{
    Book lBook1, lBook2;

    cin >> lBook1 >> lBook2;                // read books

    if ( lBook1.hasSameISBN( lBook2 ) )     // security check
    {
        cout << lBook1 + lBook2 << endl;    // write sales data
        return 0;
    }
    else
    {
        cerr << "Error, data must refer to same ISBN!" << endl;
        return 0;
    }
}
```

AddSalesContinuously



AddSalesContinuously.cpp

```cpp
int main()
{
    Book lTotal, lCurrent;

    if ( cin >> lTotal )                 // is there data to process?
        while ( cin >> lCurrent )        // continue with transactions
        {
            if ( lTotal.hasSameISBN( lCurrent ) )
                lTotal += lCurrent;
            else
            {
                cout << lTotal << endl;
                lTotal = lCurrent;
            }
            cout << lTotal << endl;
        }
    else
    {
        cerr << "Error, no data!" << endl;
        return -1;
    }
    return 0;
}
```

Line: 28 Column: 1 C++ Tab Size: 4 main

Terminal

```
Sela:HIT3303 Markus$ ./AddSalesContinuously
0-201-78345-X 2 25.00
0-201-78345-X 3 20.00
0-201-78345-X   5       110     22
0-201-78345-Y 10 15.00
0-201-78345-X   5       110     22
0-201-78345-Y   10      150     15
Sela:HIT3303 Markus$
```