A Brief Introduction to C++

Overview

- C++ Programming Model
- Coding Conventions (used in COS30008)
- A Simple Particle Simulation: A First Example
  - Object Construction
  - Operators and Member Functions
  - Class Composition
- Object-oriented Programming in C++

References

- Stanley B. Lippman, Josée Lajoie, and Barbara E. Moo: C++ Primer. 5th Edition. Addison-Wesley (2013)
- Scott Meyers: Effective Modern C++. O'Reilly (2014)
- Anthony Williams: C++ Concurrency in Action - Practical Multithreading. Manning Publications Co. (2012)

Why C++

- We need to know more than just Java or C#.
- 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, C++14, C++17, C++20,...).
- C++11 differs greatly from C++98. C++14 differs ...
- C++11 introduces move semantics and lambda expressions (revised in C++14)

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

Memory (RAM/ROM/EEPROM)

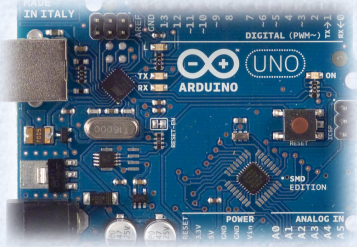
Arithmetic Logic Unit

Control Unit

Accumulator

Input

Output



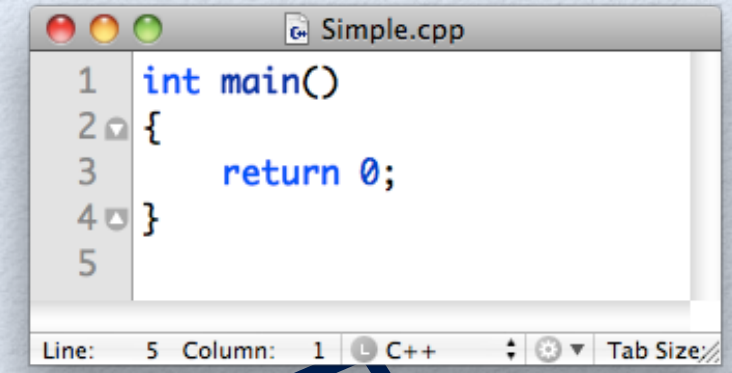
Source: http://arduino.cc/en/Main/ArduinoBoardUno

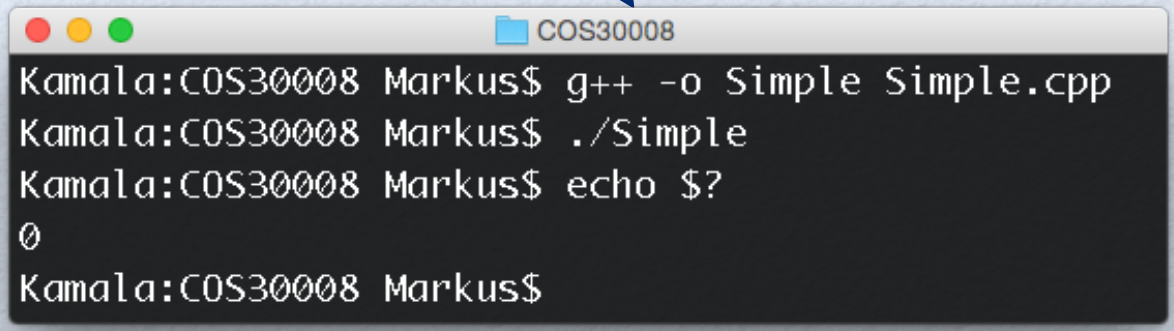
Object-Oriented Programming

- 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).

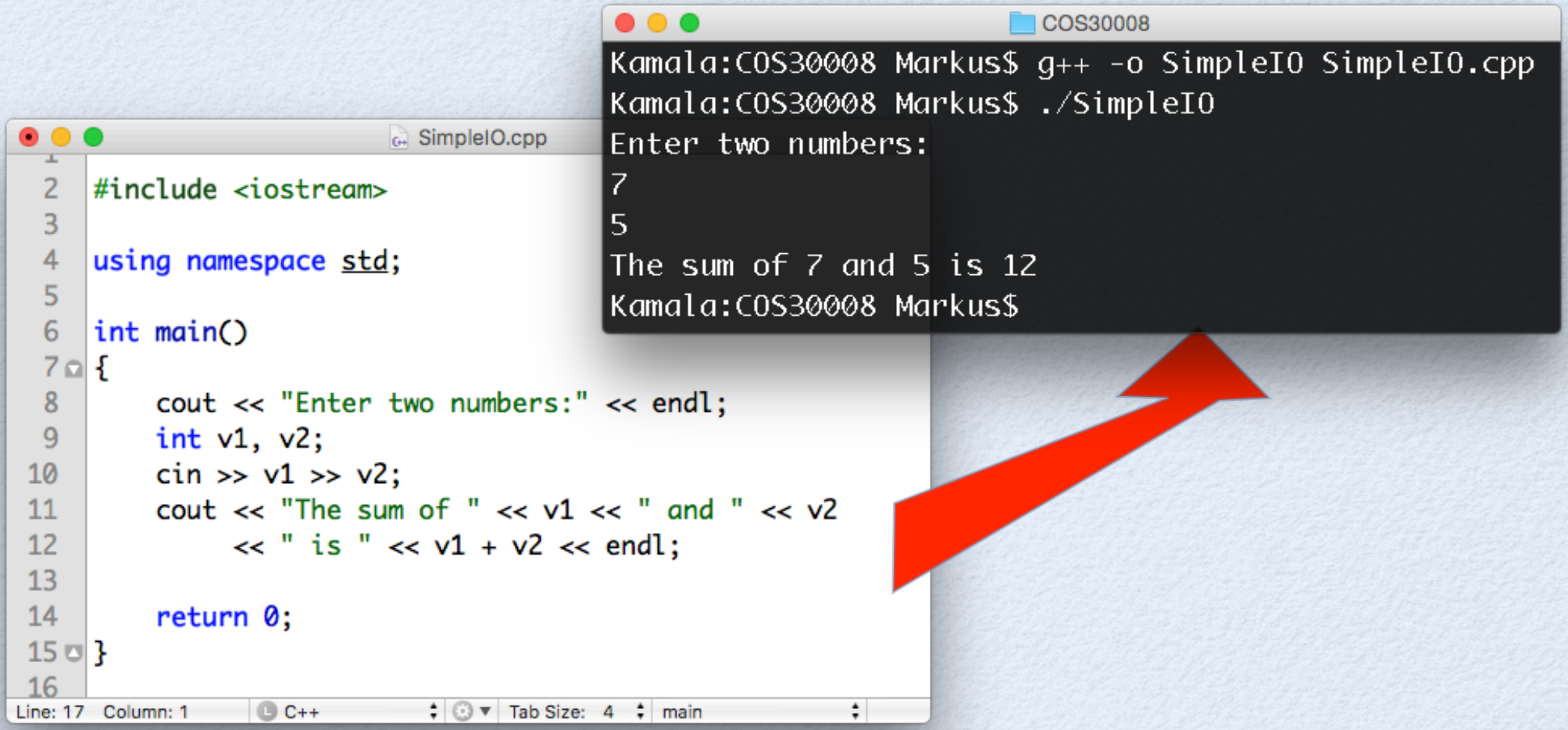
The Simplest Possible C++ Program

```
 .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
```





Let's make the program more responsive!



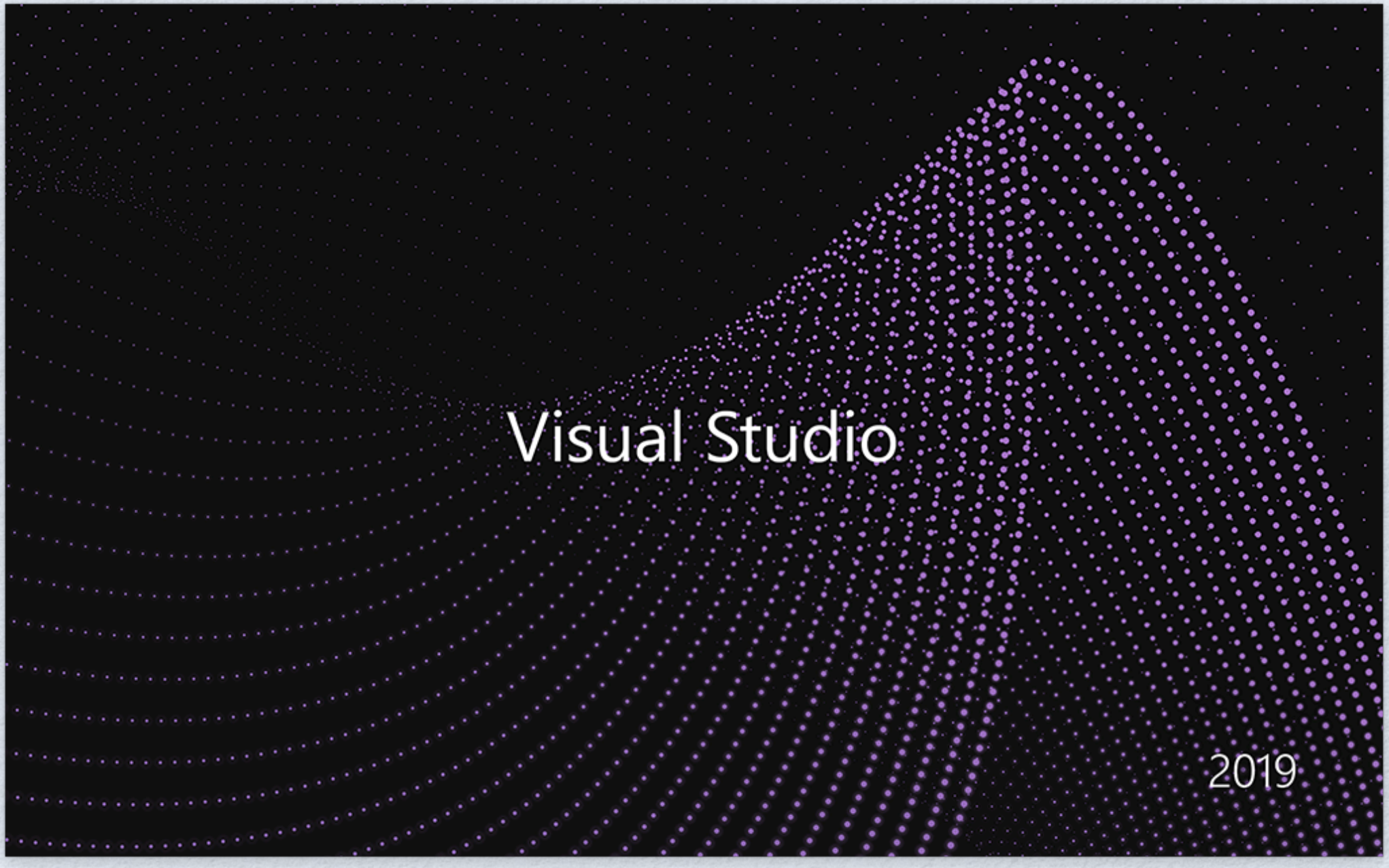
- C++ does not directly define any I/O primitives.
- I/O operations are provided by standard libraries.

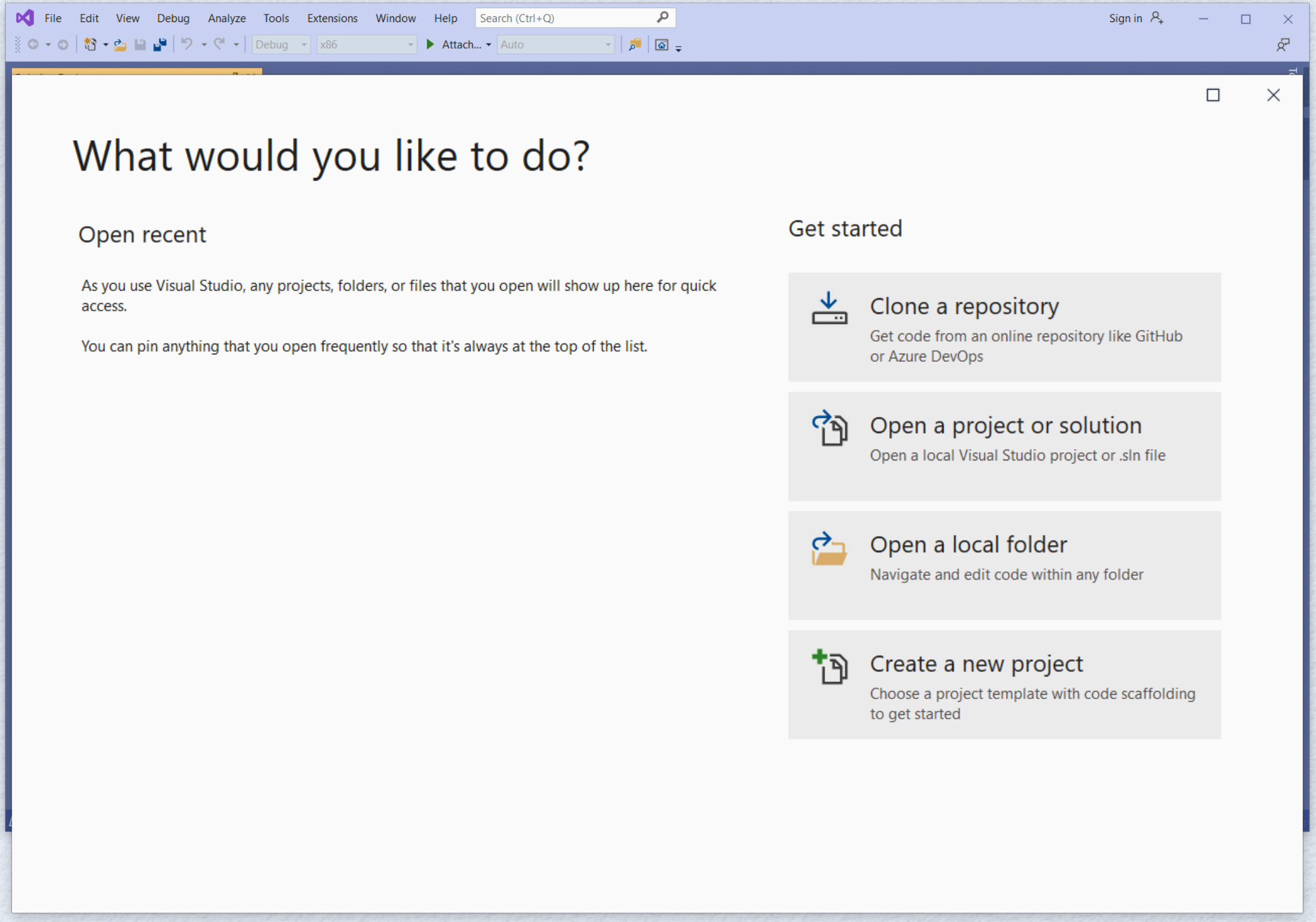
The Standard Input Stream cin

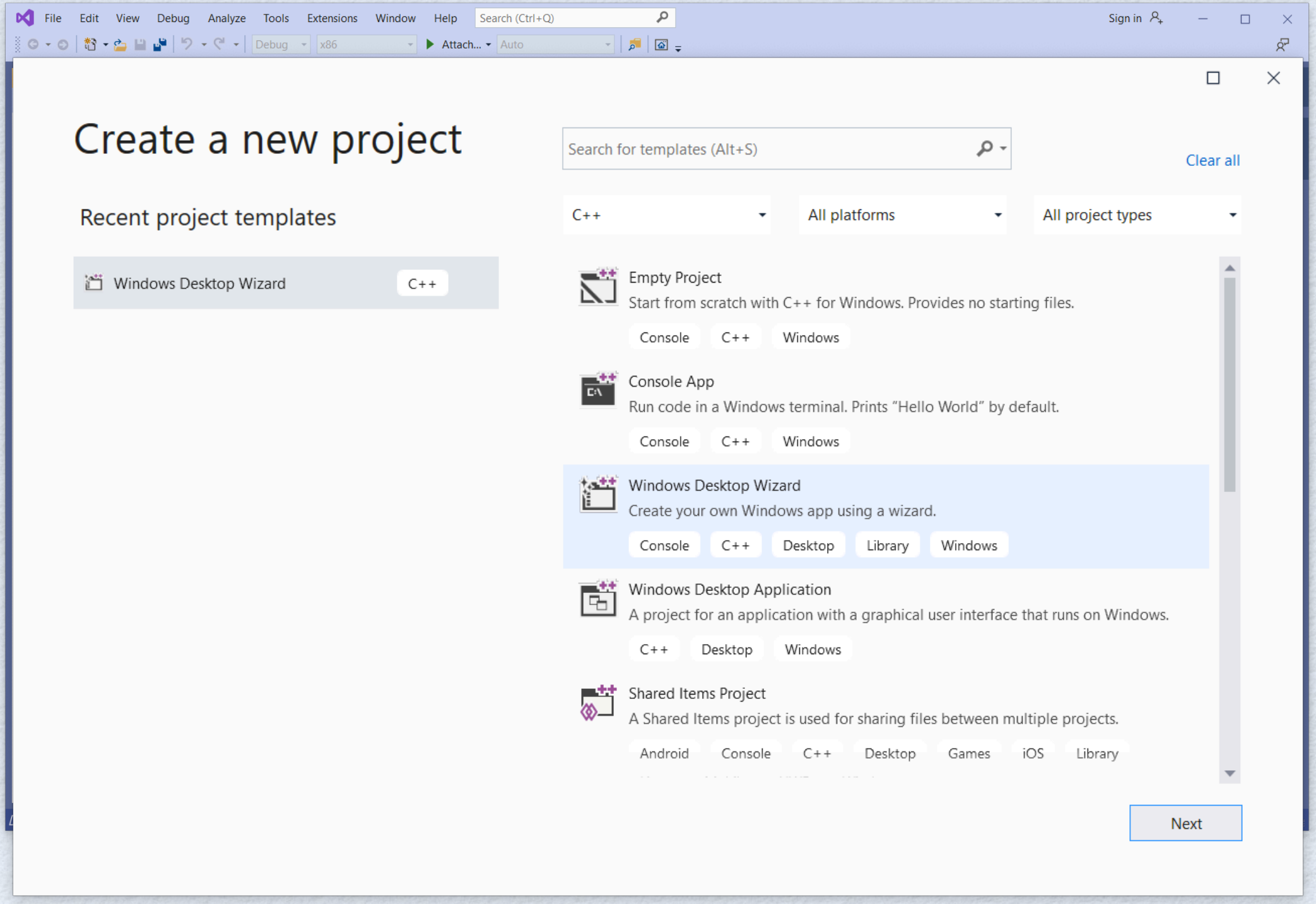
- cin is an object of class istream that represents the standard input stream. It corresponds to stdin in C.
- cin is a globally visible object that is readily available to any C++ compilation unit (i.e., a .cpp-file) that includes the library iostream.
- cin receives input either from the keyboard or a stream associated with the standard input stream.
- We use the operator >> to fetch formatted data or use the methods read or get to retrieve unformatted data from the standard input stream.

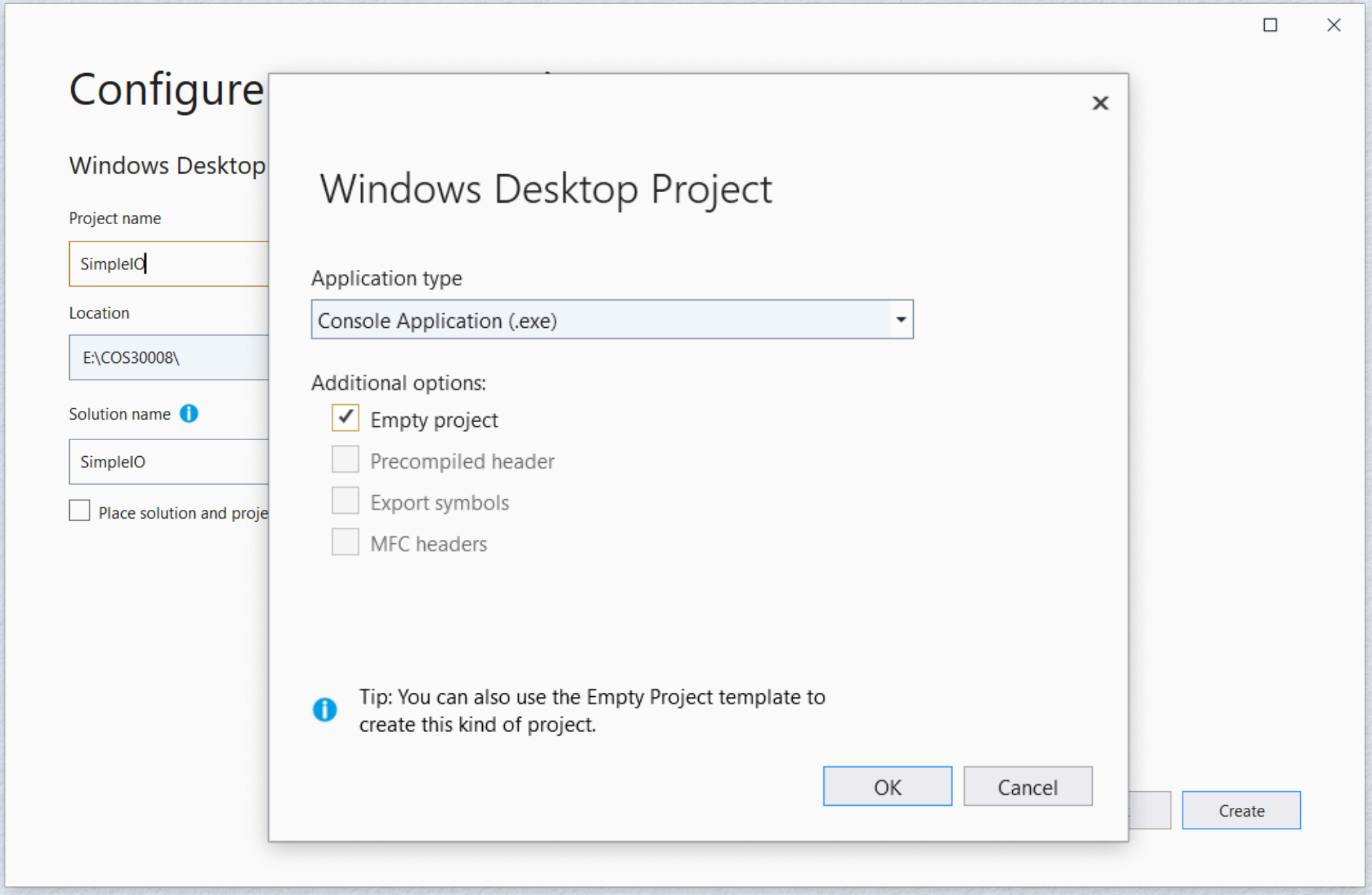
The Standard Output Stream cout

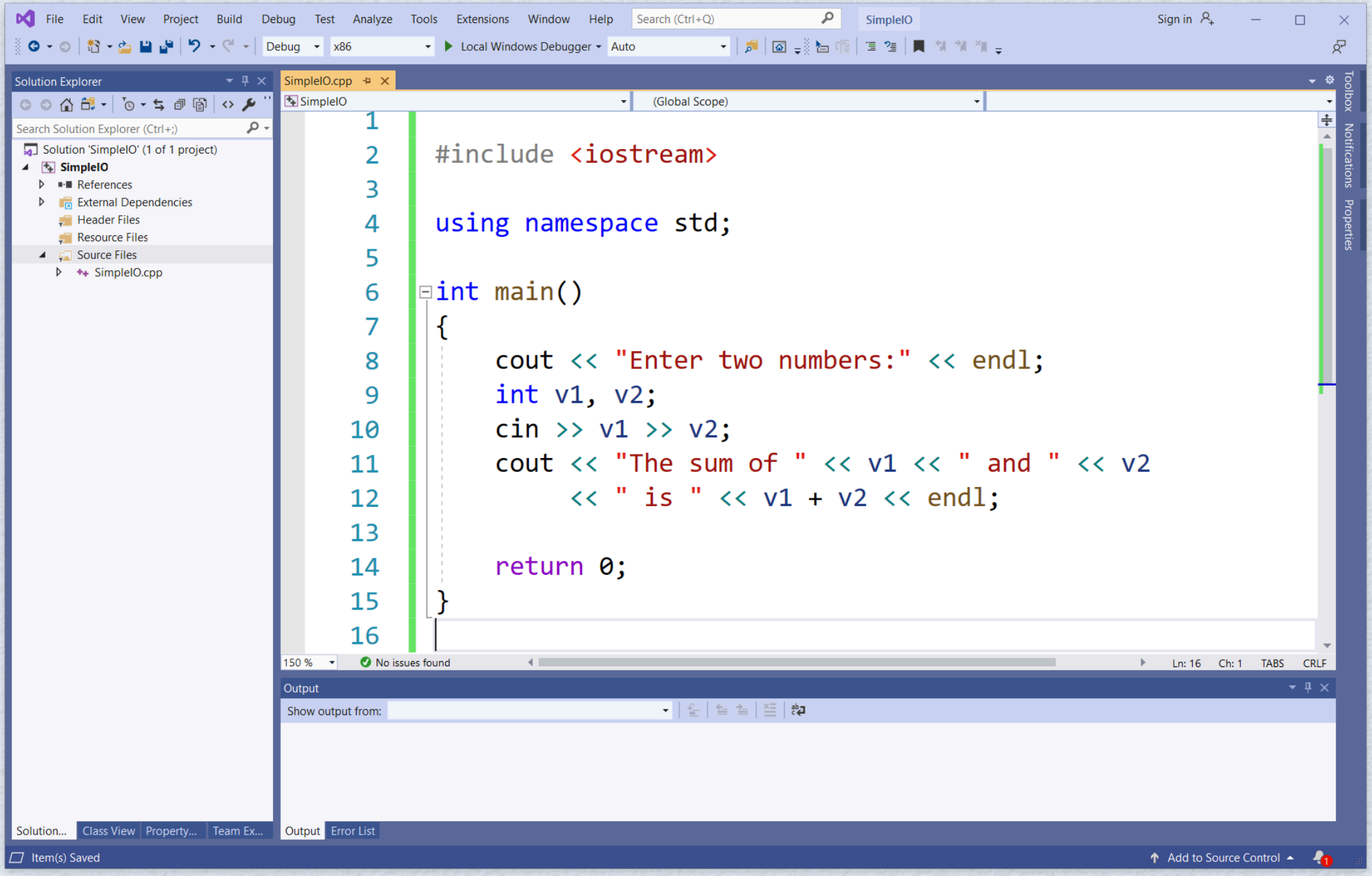
- cout is an object of class ostream that represents the standard output stream. It corresponds to stdout in C.
- cout is a globally visible object that is readily available to any C++ compilation unit (i.e., a .cpp-file) that includes the library iostream.
- cout sends data either to the console (as text) or a stream associated with the standard output stream.
- We use the operator << to push formatted data or use the methods write or put to send unformatted data to the standard output stream.

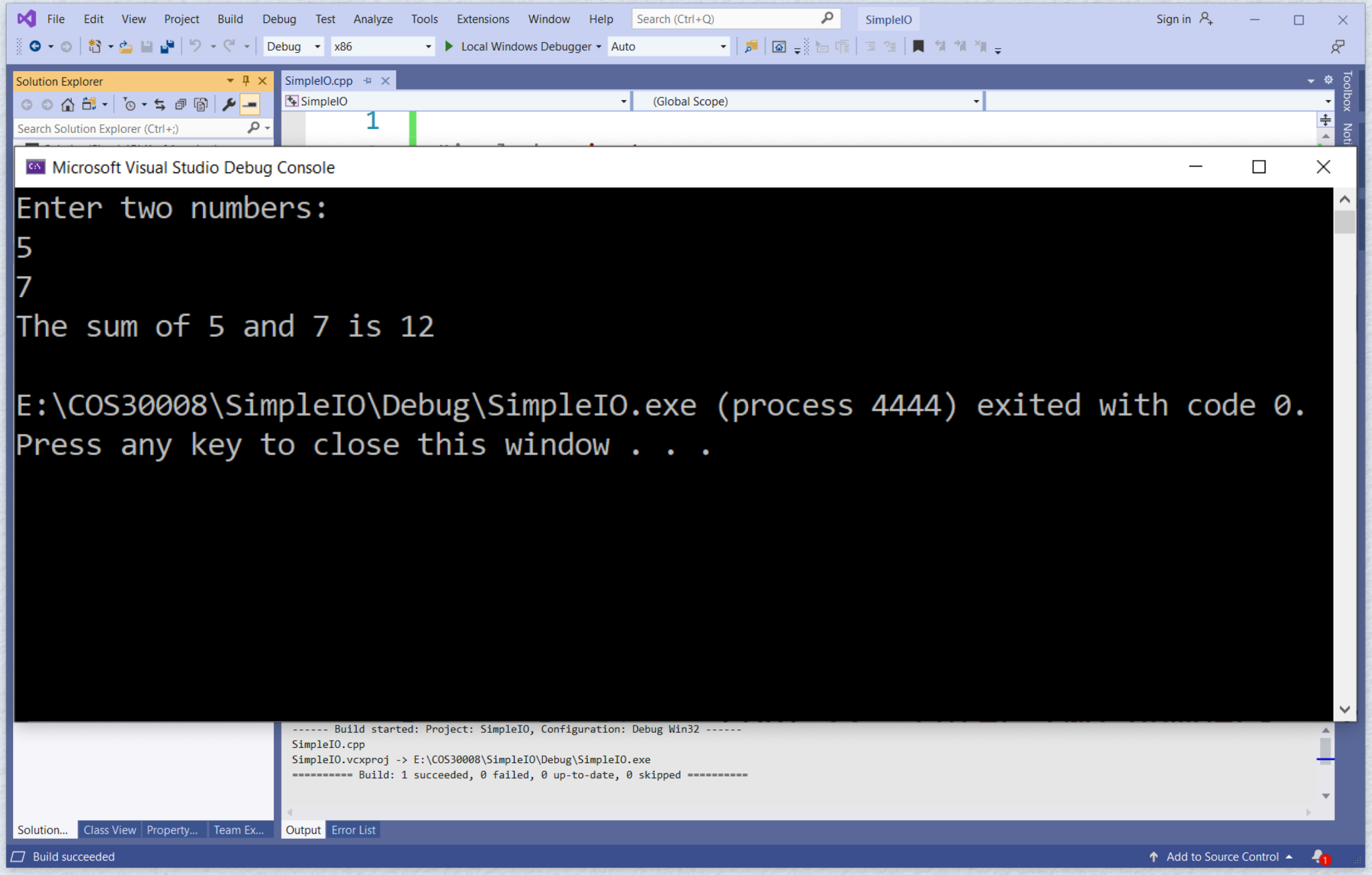












A Simple Particle Simulation

- Before we can write a program in a new language, we need to know some of its basic features. C++ is no exception.
- Let's start with a simple particle simulation (based on ideas presented on www.codingmath.com).
- The simulation program requires us to
  - define variables
  - perform input and output
  - define two data structures (i.e., classes) to hold the data we are managing: Vector2D and Particle2D
  - implement basic vector math operations for Vector2D and Particle2D
  - write control code to simulate a particle object

Technical References

- Eric Lengyel: Mathematics for 3D Game Programming and Computer Graphics, Course Technology (2012)
- Stanley B. Lippman, Josée Lajoie, and Barbara E. Moo: C++ Primer. 5th Edition. Addison-Wesley (2013)
- Kenneth H. Rosen: Discrete Mathematics and Its Application. 7th Edition. McGraw-Hill (2012)
- Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein: Introduction to Algorithms. 3rd Edition. The MIT Press (2009)
- Paul Orland: Math for Programmers - 3D graphics, machine learning, and simulations with Python. Manning Publications (2020)
- www.codingmath.com

2D Vector Operations

Length:

$$\|\vec{v}\| = \sqrt{x^2 + y^2}$$

Dot Product:

$$\vec{v} \cdot \vec{u} = v_x u_x + v_y u_v = \|\vec{v}\| \|\vec{u}\| \mathbf{cos}\ \alpha$$

Cross Product*:

$$\vec{v} \times \vec{u} = \mathbf{det} \begin{pmatrix} v_x & u_x \\ v_y & u_y \end{pmatrix} = v_x u_y - u_x v_y$$

Align Orientation:

$$\vec{v'} = \begin{bmatrix} \mathbf{cos}\ \alpha \\ \mathbf{sin}\ \alpha \end{bmatrix} \|\vec{v}\|$$

Orientation/Direction:

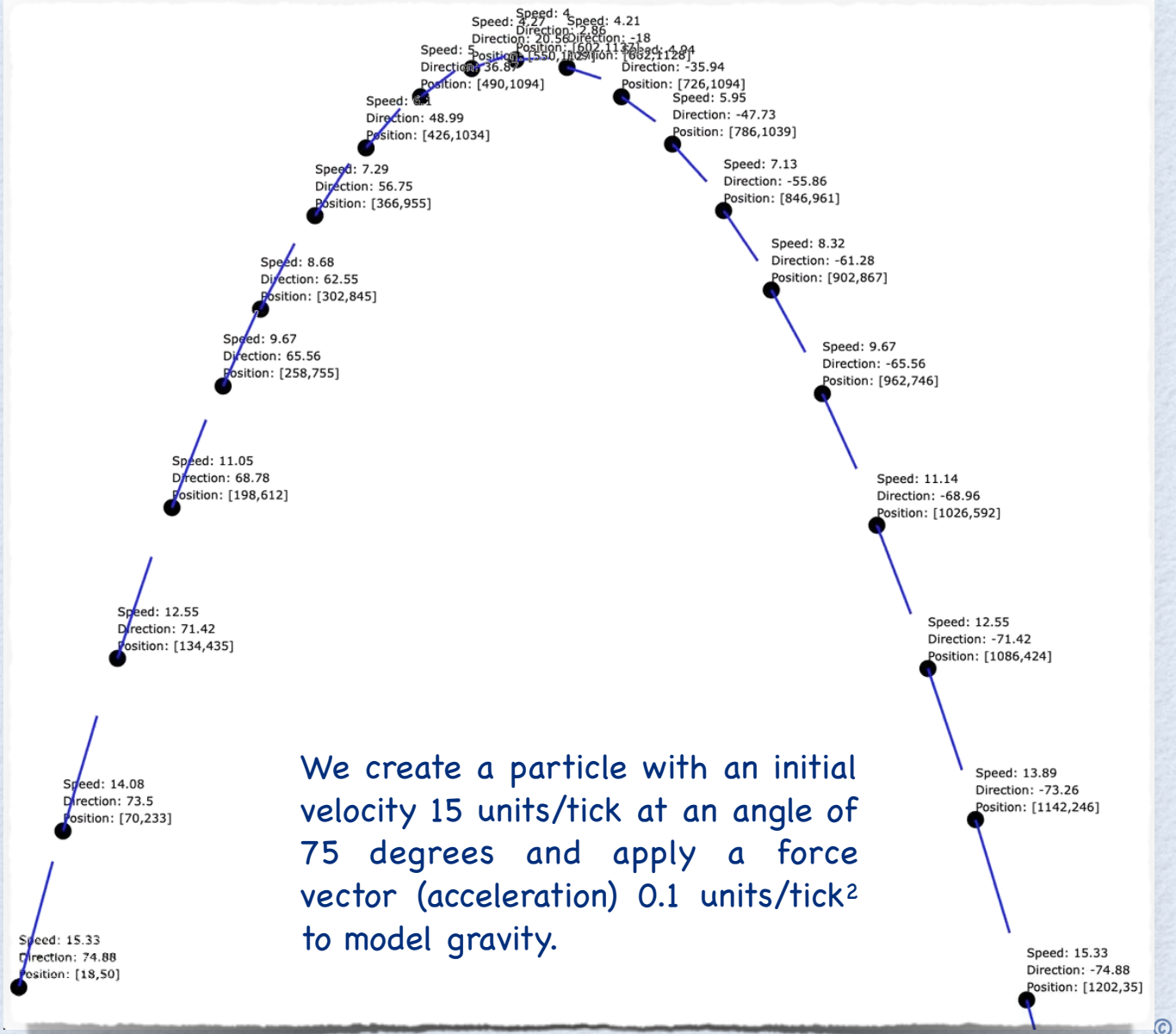
$$\theta = \mathbf{tan}^{-1}(v_y / v_x)$$

Normalization:

$$\vec{n_v} = \frac{\vec{v}}{\|\vec{v}\|}, \|\vec{n_v}\| = 1$$

* The cross product is a three-dimensional concept. In 2D, we use it to determine wether consecutive line segments turn left of right.

Visualization of the Particle Simulation



We create a particle with an initial velocity 15 units/tick at an angle of 75 degrees and apply a force vector (acceleration) 0.1 units/tick^2 to model gravity.

Preliminaries
Coding Conventions

Coding Conventions

- Coding conventions establish guidelines for a specific programming language that recommend programming style, practices, and methods for each aspect of a program written in this language.
- Coding conventions are only applicable to the human maintainers and peer reviewers of a software project.
- Conventions may be formalized in a documented set of rules that an entire team or company follows, or may be as informal as the habitual coding practices of an individual.
- Coding conventions are not enforced by compilers.

CamelCase

- CamelCase (or medial capitals) is a practice of writing names of variable or functions with some inner uppercase letters to denote embedded words:
  - InputStreamReader: character input stream
  - getEncoding: getter method for data encoding
  - MyIntegerArray: array variable
  - CreateWindowEx: Windows function
- Two popular variants:
  - Pascal InfixCaps - the first letter should be a capital, and any embedded words in an identifier should be in caps, as well as any acronym that is embedded.
  - Java - variables are mixed case with a lowercase first letter, methods should be verbs, in mixed case with the first letter lowercase, and classes should be nouns, in mixed case with the first letter of each internal word capitalized.

Borland-Inspired Style Guide Elements

- Field names should start with the letter 'f'.
- Local variable names should start with the letter 'l'.
- Parameter names should start with the letter 'a'.
- Global variable names should start with the letter 'g'.

Which Coding Standard To Use

- Coding conventions are not enforced by compilers.
- Not following some or all of the rules has no impact on the executable programs created from the source code.
- Code standards facilitate program comprehension and make program maintenance easier.
- Every organization may enforce some sort of coding standards as part of organization's quality assurance process.

Vector2D: A Basic 2D Vector Class

Vector2D.h — Introduction

```
4  #pragma once
5
6  #include <iostream>
7
8  class Vector2D
9  {
10 private:
11
12     float fX;
13     float fY;
14
15 public:
16
17     Vector2D( float aX = 1.0f, float aY = 0.0f ) : fX(aX), fY(aY) {}
18     Vector2D( std::istream& aIStream ) { aIStream >> *this; }
19
20     float getX() const;
21     float getY() const;
22
23     Vector2D operator+( const Vector2D& aVector ) const;
24     Vector2D operator-( const Vector2D& aVector ) const;
25
26     Vector2D operator*( const float aScalar ) const;
27     float dot( const Vector2D& aVector ) const;
28     float cross( const Vector2D& aVector ) const;
29
30     float length() const;
31     Vector2D normalize() const;
32
33     float direction() const;
34     Vector2D align( float aAngleInDegrees ) const;
35
36     friend std::istream& operator>>( std::istream& aIStream, Vector2D& aVector );
37     friend std::ostream& operator<<( std::ostream& aOStream, const Vector2D& aVector );
38 };
39
40 Vector2D operator*( const float aScalar, const Vector2D& aVector );
```

Line: 2:16 C++ Tab Size: 4

The Structure of a Class

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

Don't 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.

Vector2D: Private Members

Vector2D.h — Introduction

```
4   #pragma once
5
6   #include <iostream>
7
8   class Vector2D
9   {
10  private:
11
12      float fX;
13      float fY;
14
15  public:
16
17      Vector2D( float aX = 1.0f, float aY = 0.0f ) : fX(aX), fY(aY) {}
18      Vector2D( std::istream& aIStream ) { aIStream >> *this; }
19
20      float getX() const;
21      float getY() const;
22
23      Vector2D operator+( const Vector2D& aVector ) const;
24      Vector2D operator-( const Vector2D& aVector ) const;
25
26      Vector2D operator*( const float aScalar ) const;
27      float dot( const Vector2D& aVector ) const;
28      float cross( const Vector2D& aVector ) const;
29
30      float length() const;
31      Vector2D normalize() const;
32
33      float direction() const;
34      Vector2D align( float aAngleInDegrees ) const;
35
36      friend std::istream& operator>>( std::istream& aIStream, Vector2D& aVector );
37      friend std::ostream& operator<<( std::ostream& aOStream, const Vector2D& aVector );
38  };
39
40  Vector2D operator*( const float aScalar, const Vector2D& aVector );
```

Line: 2:16 C++ Tab Size: 4

vector coordinates

Vector2D: Public Members

Vector2D.h — Introduction

```
4  #pragma once
5
6  #include <iostream>
7
8  class Vector2D
9  {
10 private:
11
12     float fX;
13     float fY;
14
15 public:
16
17     Vector2D( float aX = 1.0f, float aY = 0.0f ) : fX(aX), fY(aY) {}
18     Vector2D( std::istream& aIStream ) { aIStream >> *this; }
19
20     float getX() const;
21     float getY() const;
22
23     Vector2D operator+( const Vector2D& aVector ) const;
24     Vector2D operator-( const Vector2D& aVector ) const;
25
26     Vector2D operator*( const float aScalar ) const;
27     float dot( const Vector2D& aVector ) const;
28     float cross( const Vector2D& aVector ) const;
29
30     float length() const;
31     Vector2D normalize() const;
32
33     float direction() const;
34     Vector2D align( float aAngleInDegrees ) const;
35
36     friend std::istream& operator>>( std::istream& aIStream, Vector2D& aVector );
37     friend std::ostream& operator<<( std::ostream& aOStream, const Vector2D& aVector );
38 };
39
40 Vector2D operator*( const float aScalar, const Vector2D& aVector );
```

Line: 2:16 C++ Tab Size: 4

Basic vector operations

Vector2D: Friends

Vector2D.h — Introduction

```
4   #pragma once
5
6   #include <iostream>
7
8   class Vector2D
9   {
10  private:
11
12      float fX;
13      float fY;
14
15  public:
16
17      Vector2D( float aX = 1.0f, float aY = 0.0f ) : fX(aX), fY(aY) {}
18      Vector2D( std::istream& aIStream ) { aIStream >> *this; }
19
20      float getX() const;
21      float getY() const;
22
23      Vector2D operator+( const Vector2D& aVector ) const;
24      Vector2D operator-( const Vector2D& aVector ) const;
25
26      Vector2D operator*( const float aScalar ) const;
27      float dot( const Vector2D& aVector ) const;
28      float cross( const Vector2D& aVector ) const;
29
30      float length() const;
31      Vector2D normalize() const;
32
33      float direction() const;
34      Vector2D align( float aAngleInDegrees ) const;
35
36      friend std::istream& operator>>( std::istream& aIStream, Vector2D& aVector );
37      friend std::ostream& operator<<( std::ostream& aOStream, const Vector2D& aVector );
38  };
39
40  Vector2D operator*( const float aScalar, const Vector2D& aVector );
```

Line: 2:16 C++ Tab Size: 4

Support for C++ stream-based I/O

Vector2D: Ad hoc Definitions

Vector2D.h — Introduction

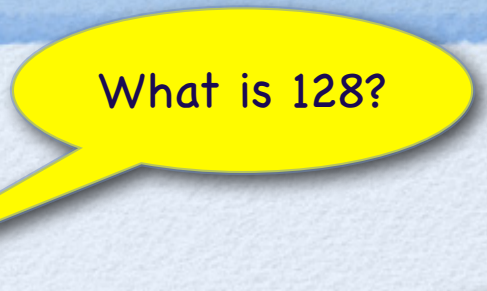
```
4  #pragma once
5
6  #include <iostream>
7
8  class Vector2D
9  {
10 private:
11
12     float fX;
13     float fY;
14
15 public:
16
17     Vector2D( float aX = 1.0f, float aY = 0.0f ) : fX(aX), fY(aY) {}
18     Vector2D( std::istream& aIStream ) { aIStream >> *this; }
19
20     float getX() const;
21     float getY() const;
22
23     Vector2D operator+( const Vector2D& aVector ) const;
24     Vector2D operator-( const Vector2D& aVector ) const;
25
26     Vector2D operator*( const float aScalar ) const;
27     float dot( const Vector2D& aVector ) const;
28     float cross( const Vector2D& aVector ) const;
29
30     float length() const;
31     Vector2D normalize() const;
32
33     float direction() const;
34     Vector2D align( float aAngleInDegrees ) const;
35
36     friend std::istream& operator>>( std::istream& aIStr   , Vector2D& aVector );
37     friend std::ostream& operator<<( std::ostream& aOS   eam, const Vector2D& aVector );
38 };
39
40 Vector2D operator*( const float aScalar, const Vector2D& aVector );
```

Line: 2:16 C++ Tab Size: 4

Support for scalar * vector

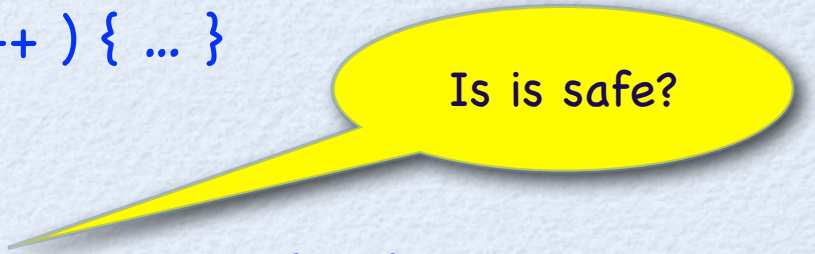
const

Constants and Const Qualifier



- What are the problems with

```
for ( int index = 0; index < 128; index++ ) { ... }
```



- We can do better

```
for ( int index = 0; index < BufferSize; index++ ) { ... }
```

- Defining a const object:

```
const int BufferSize = 128;          // initialized at compile time
```

- Unlike macro definitions, const objects have an address!

```
#define BUF_SIZE 128                 // macro definition
const int BufferSize = BUF_SIZE;     // initialized at compile time
```

Everything that should or must not change is marked with the **const** keyword.

Initialized Vector2D objects are read-only.

Vector2D.h — Introduction

```
4  #pragma once
5
6  #include <iostream>
7
8  class Vector2D
9  {
10 private:
11
12     float fX;
13     float fY;
14
15 public:
16
17     Vector2D( float aX = 1.0f, float aY = 0.0f ) : fX(aX), fY(aY) {}
18     Vector2D( std::istream& aIStream ) { aIStream >> *this; }
19
20     float getX() const;
21     float getY() const;
22
23     Vector2D operator+( const Vector2D& aVector ) const;
24     Vector2D operator-( const Vector2D& aVector ) const;
25
26     Vector2D operator*( const float aScalar ) const;
27     float dot( const Vector2D& aVector ) const;
28     float cross( const Vector2D& aVector ) const;
29
30     float length() const;
31     Vector2D normalize() const;
32
33     float direction() const;
34     Vector2D align( float aAngleInDegrees ) const;
35
36     friend std::istream& operator>>( std::istream& aIStream, Vector2D& aVector );
37     friend std::ostream& operator<<( std::ostream& aOStream, const Vector2D& aVector );
38 };
39
40 Vector2D operator*( const float aScalar, const Vector2D& aVector );
```

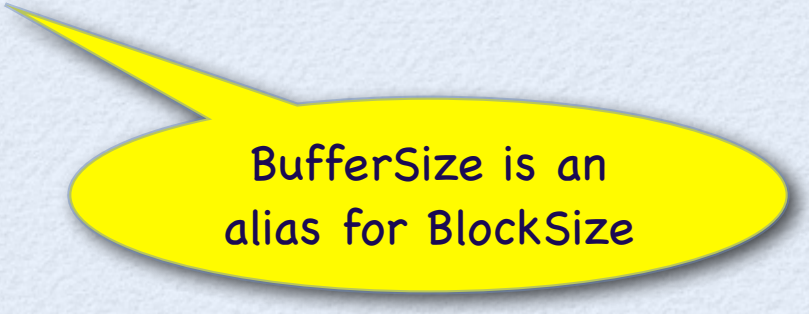
Line: 2:16 C++ Tab Size: 4

const member functions

References (C++-98)

- A reference introduces a new name (alias) for an existing object:

```
int BlockSize = 512;

int& BufferSize = BlockSize;
```



Constant References (C++-98)

- A constant reference yields a new name for a constant object:

```
const int FixedBlockSize = 512;

const int& FixedBufferSize = FixedBlockSize;
```

- A constant reference defines an immutable alias to an existing object.

References prevent copies from being made.

Reference Parameters (C++-98)

- C++ uses call-by-value as default parameter passing mechanism.

```
void Assign( int aPar, int aVal ) { aPar = aVal; }

Assign( val, 3 );                    // val unchanged
```

- A reference parameter yields call-by-reference:

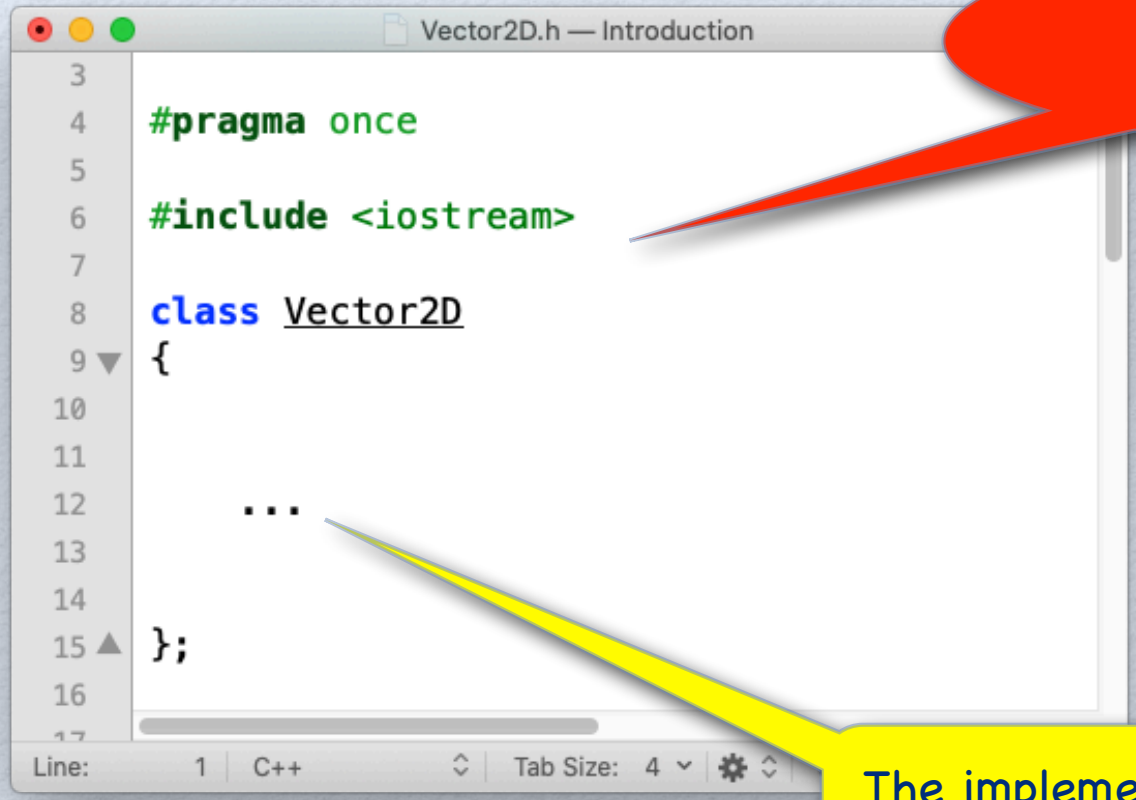
```
void AssignR( int& aPar, int aVal ) { aPar = aVal; }

AssignR( val, 3 );                   // val is set to 3
```

- A const reference parameter yields call-by-reference, but the value of the parameter is read-only:

```
void AssignCR( const int& aPar, int aVal ) { aPar = aVal; } // error
```

Class Implementation

Include File: Vector2D.h

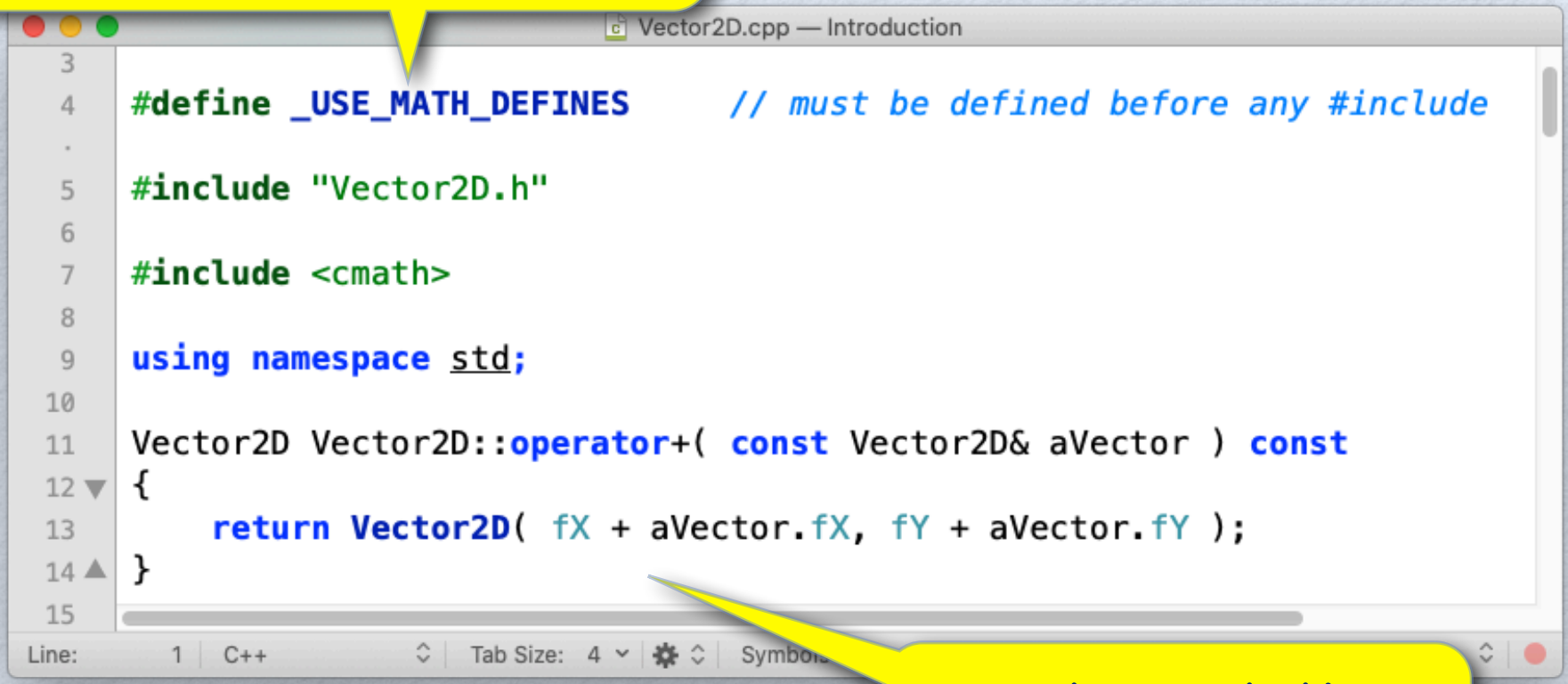


We do not select any namespace yet!

The implementation goes to Vector2D.cpp

Implementation File: Vector2D.cpp

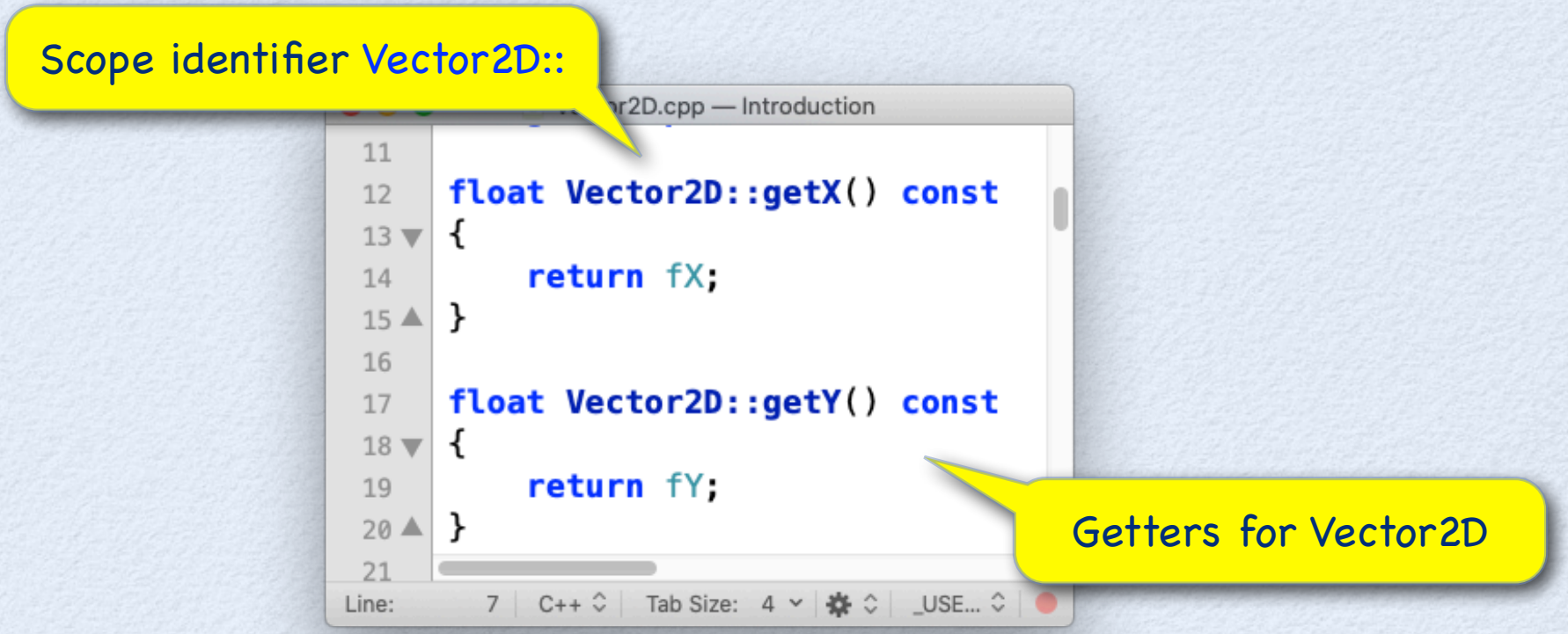
Macro definition to make math definitions available (e.g., M_PI)



Implementation

Member Implementation

- When implementing a member function of a class in C++ we must explicitly specify the class name using a scope identifier within the signature of the member function.
- A scope identifier is a name followed by two colons (e.g. Vector2D::).



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 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

Object Initialization

- A class defines a new data type. Instances of this data type are objects that need initialization.
- Each class defines explicitly (or implicitly) some special member functions, called constructors, that are executed whenever we create new objects of a class.
- The job of a constructor is to ensure that the data members of each objects are set to some sensible initial values.

Constructors

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

Default Constructor

- A default constructor is one that does not take any arguments.
- The compiler will synthesize a default constructor, when no other constructors have been specified.
- There are situations when the compiler needs to create objects in an environment agnostic way (e.g., array of objects). In this case the compiler relies on the existence of the default constructor.
- If some data members have built-in or compound types, then the class should not rely on the synthesized default constructor!

Class Vector2D - Constructors

Vector2D.h — Introduction

```
4  #pragma once
5
6  #include <iostream>
7
8  class Vector2D
9  {
10 private:
11
12     float fX;
13     float fY;
14
15 public:
16
17     Vector2D( float aX = 1.0f, float aY = 0.0f ) : fX(aX), fY(aY) {}
18     Vector2D( std::istream& aIStream ) { aIStream >> *this; }
19
20     float getX() const;
21     float getY() const;
22
23     Vector2D operator+( const Vector2D& aVector ) const;
24     Vector2D operator-( const Vector2D& aVector ) const;
25
26     Vector2D operator*( const float aScalar ) const;
27     float dot( const Vector2D& aVector ) const;
28     float cross( const Vector2D& aVector ) const;
29
30     float length() const;
31     Vector2D normalize() const;
32
33     float direction() const;
34     Vector2D align( float aAngleInDegrees ) const;
35
36     friend std::istream& operator>>( std::istream& aIStream, Vector2D& aVector );
37     friend std::ostream& operator<<( std::ostream& aOStream, const Vector2D& aVector );
38 };
39
40 Vector2D operator*( const float aScalar, const Vector2D& aVector );
```

Line: 2:16 C++ Tab Size: 4

Default argument

Note, we use inlined trivial constructors here.

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.
- Constructor initializers take the form of function calls where the name of the function coincides with name of the instance variable being initialized.

```
Vector2D( float aX = 1.0f, float aY = 0.0f ) : fX(aX), fY(aY) {}
```

Implicit Class-Type Conversions

- A constructor that can be called with a single argument defines an implicit conversion from the parameter type to the class type.

```
Vector2D( std::istream& aIStream ) { aIStream >> *this; }
```

Conversion from std::istream to Vector2D

- Accordingly, we can use an istream where an object of type Vector2D is expected:

```
Vector2D( std::istream& aIStream ) { aIStream >> *this; }

Vector2D operator+( const Vector2D& aVector ) const;
```

```
Vector2D result = vec + cin;
```

std::istream -> Vector2D

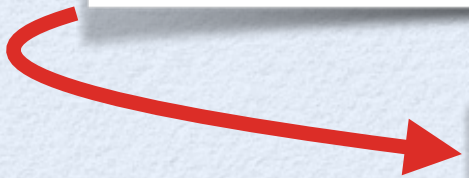
Suppressing Implicit Conversions

- When a constructor is declared explicit, the compiler will not use it as a conversion operator.

```
explicit Vector2D( std::istream& aIStream ) { aIStream >> *this; }
```

- Accordingly, we cannot use an istream where an object of type Vector2D is expected:

```
explicit Vector2D( std::istream& aIStream ) { aIStream >> *this; }

Vector2D operator+( const Vector2D& aVector ) const;
```



```
Vector2D result = vec + cin;
```

Error

It is application-dependent whether implicit conversion needs to be supported.

Rule of Thumb

- Keep C++ headers free of implementations, unless you want the implementations to be inlined or they are templates.

Basic 2D vector operations

Operator Overloading

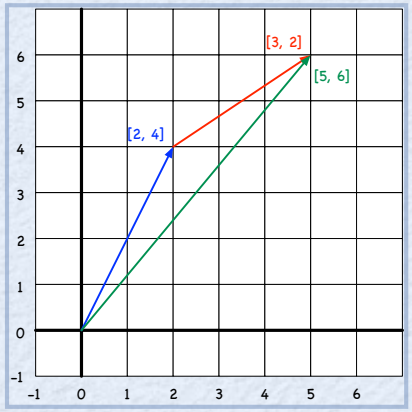
- C++ supports operator overloading.
- Overloaded operators are like normal functions, but are defined using a pre-defined operator symbol.
- You cannot change the priority and associativity of an operator.
- Operators are selected by the compiler based on the static types of the specified operands.

Member Operators vs Ad hoc Operators

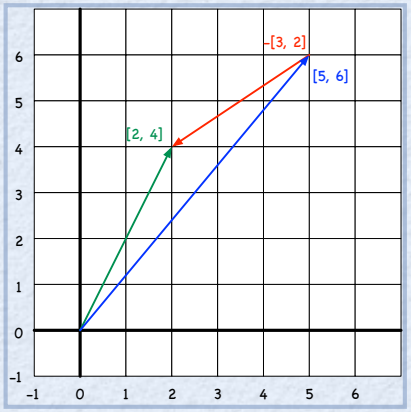
- There are two forms of operator overloading:
  - member operator
  - ad hoc operator
- A member operator receives a first implicit argument - this object. In other words, an overloaded operator in C++ is like any other non-static method function defined for a class. Non-static member functions receive as first argument "this" object. As a consequence, we only specify the second argument for binary operators like '+' or '-'.
- Ad hoc operators are not members of a class. Their signature has to match the signature of the operator to be defined. That is, an ad hoc definition of a binary operator '+' requires two arguments: the left-hand side of '+' and the right-hand side of '+'.

Addition & Subtraction

[5,6] = [2,4] + [3,2]



[2,4] = [5,6] - [3,2]



Vector2D.cpp — Introduction

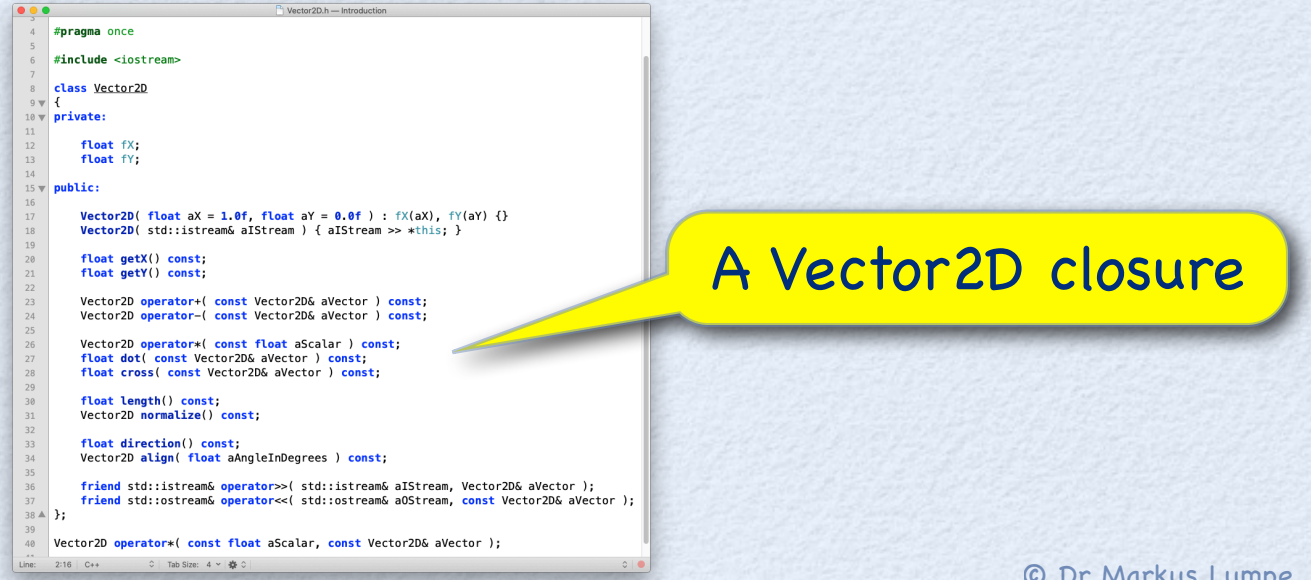
```
22  Vector2D Vector2D::operator+( const Vector2D& aVector ) const
23  {
24      return Vector2D( fX + aVector.fX, fY + aVector.fY );
25  }
26
27  Vector2D Vector2D::operator-( const Vector2D& aVector ) const
28  {
29      return Vector2D( fX - aVector.fX, fY - aVector.fY );
30  }
31
```

Tab Size: 4 _USE_MATH_DEFINES

Member operators

Closures - Lexical Scoped Name Binding

- Classes provide a lexical scoped name binding for methods. This allows methods of a class to access instance variables and other methods of the class that are defined outside the scope of a given method.
- Operationally, classes create closures that are records storing a function (or a set of functions) together with an environment. The environment associates each free variable of the function(s) to values to which the name was bound when the closure was created.



Scalar Multiplication, Dot Product, and Cross Product

- We use scalar multiplication to scale a vector uniformly.
- The dot product (inner product) is a measure of the difference between the directions in which the two vectors point.
- The 2D cross product yields a scalar that we use it to determine wether consecutive line segments turn left of right.

Vector2D.cpp — Introduction

```
31
32  Vector2D Vector2D::operator*( const float aScalar ) const
33  {
34      return Vector2D( fX * aScalar, fY * aScalar );
35  }
36
37  float Vector2D::dot( const Vector2D& aVector ) const
38  {
39      return fX * aVector.fX + fY * aVector.fY;
40  }
41
42  float Vector2D::cross( const Vector2D& aVector ) const
43  {
44      return fX * aVector.fY - fY * aVector.fX;
45  }
46
```

Line: 7 C++ Tab Size: 4 _USE_MATH_DEFINES

Vector Length and Unit Vector of a Vector

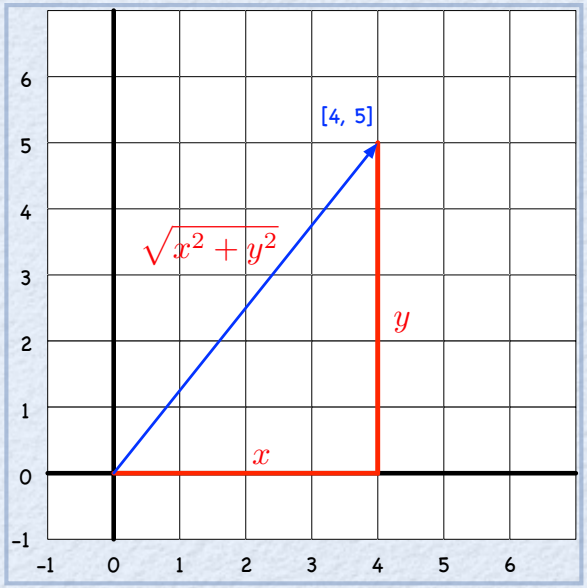
- The length of a vector (magnitude) is the hypotenuse of the right-angled triangle formed by the vector coordinates x and y.
- The unit vector of a vector is a vector with length 1. (In the code below *this refers to the this object, that is, the vector object for which we calculate the unit vector.)

Vector2D.cpp — Introduction

```
47  float Vector2D::length() const
48  {
49      float val = sqrt(fX * fX + fY * fY);
50
51      return round( val * 100.0f ) / 100.0f;
52  }
53
54  Vector2D Vector2D::normalize() const
55  {
56      return *this * (1.0f/length());
57  }
58
```

Line: 11 C++ Tab Size: 4 std

vector length



Direction and Align

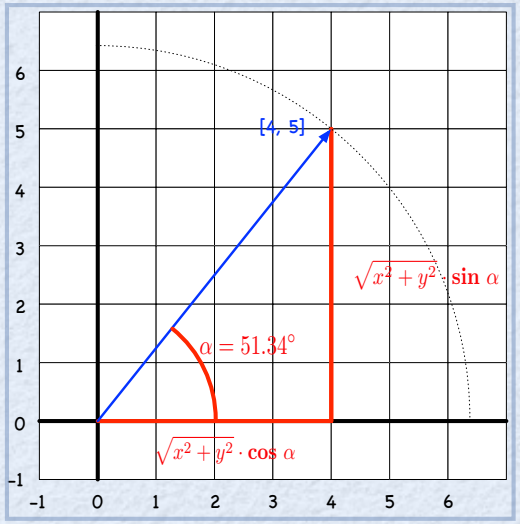
- The direction of a vector is the arctangent of the right-angled triangle formed by the vector coordinates x and y.
- We can align/rotate a vector, without changing its length, by multiplying its length with the sine and the cosine of the direction angle to obtain the new x and y coordinates, respectively.

C++ cast M_PI to float

```
59  float Vector2D::direction() const
60  {
61      float val = atan2( fY, fX ) * 180.0f / static_cast<float>(M_PI);
62
63      return round( val * 100.0f ) / 100.0f;
64  }
65
66  Vector2D Vector2D::align( float aAngleInDegrees ) const
67  {
68      float lRadians = aAngleInDegrees * static_cast<float>(M_PI) / 180.0f;
69
70      return length() * Vector2D( cos( lRadians ), sin( lRadians ) );
71  }
72
```

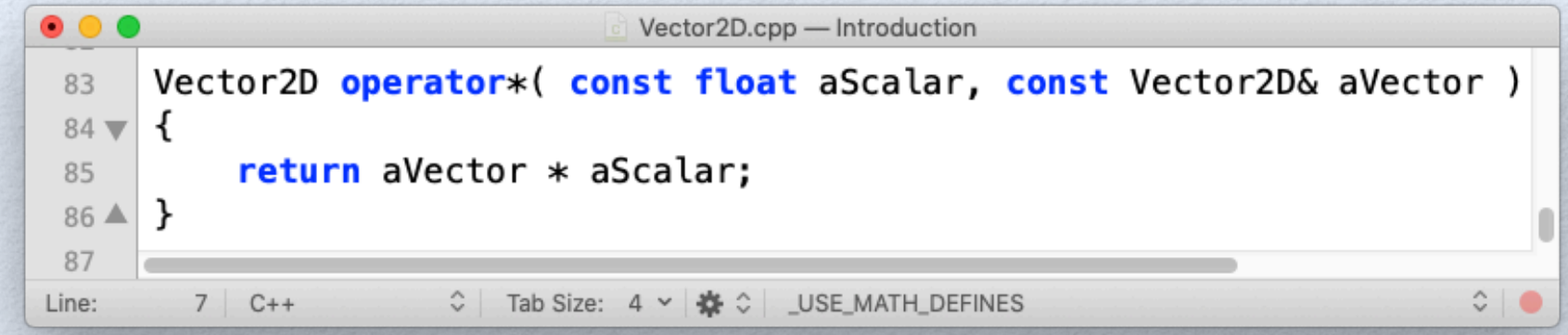
Line: 11 C++ Tab Size: 4 std

direction/align



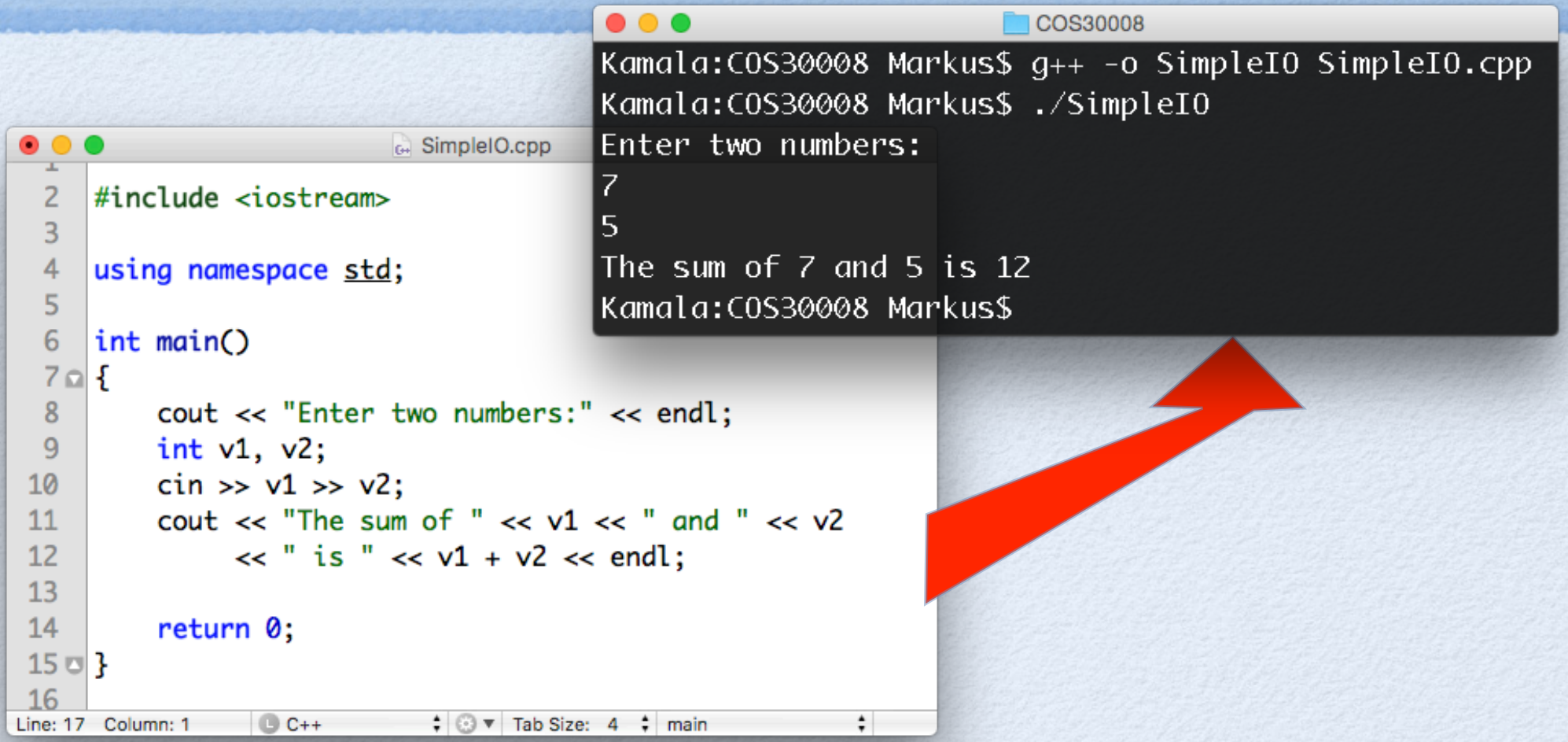
Ad hoc Operator *

- The member operator for scalar multiplication only allows for vector * scalar. However, multiplication is commutative, that is changing the order of the operands does not change the result.
- We can recover the commutativity of scalar multiplication by defining an ad hoc multiplication operator that takes a scalar as first argument and a vector as the second:



Data I/O

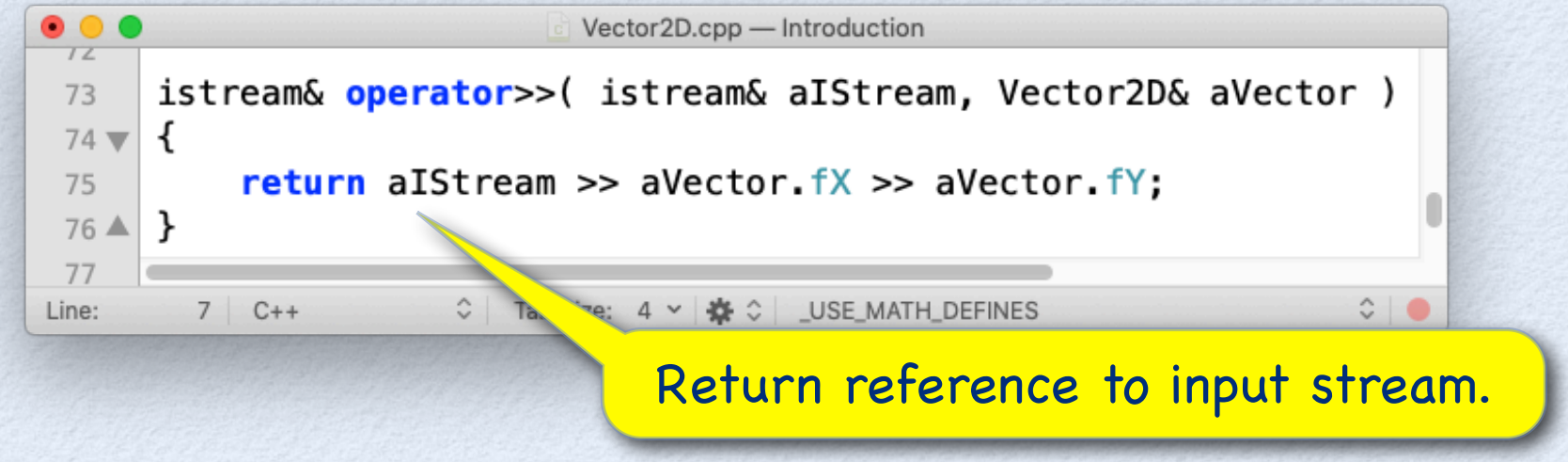
I/O in C++ is operator based.



- C++ relies on the binary operators << and >> to perform formatted input and output, respectively.

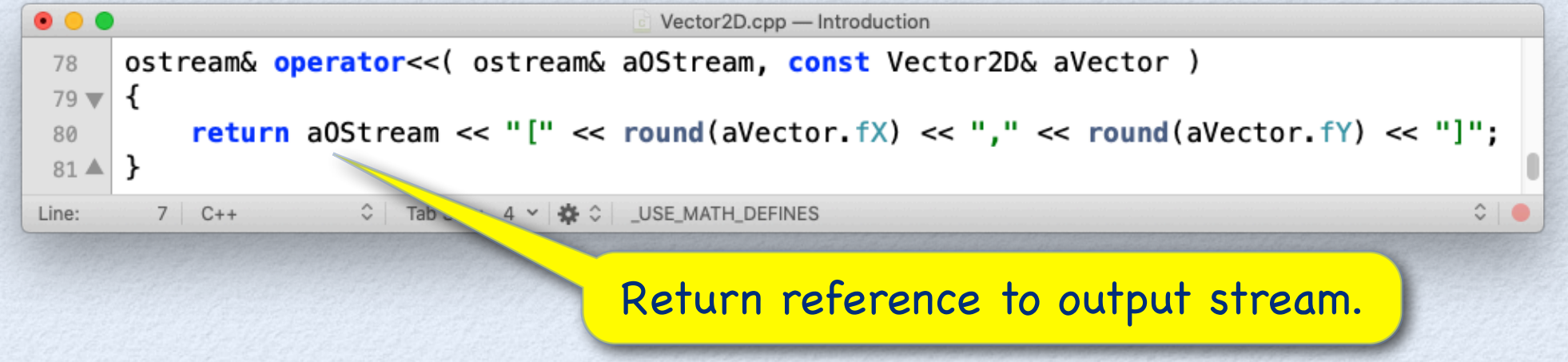
The Vector2D Input Operator >>

- The input operator performs formatted input of the vector coordinates, that is, we try to fetch two floating point values from the input stream.
- If the input fails, either of the coordinates or both are set to 0.0.
- At the end, the input operator returns the stream object to allow for a "chaining" of input operations.



The Vector2D Output Operator <<

- The output operator assigns a "textual representation" to objects of type Vector2D.
- It performs formatted output, that is, the coordinate values are rendered as integer values in the output. The round function yields the nearest integral value.
- At the end, the input operator returns the stream object to allow for a "chaining" of output operations.



Class Vector2D - The Friends

Vector2D.h — Introduction

```
4  #pragma once
5
6  #include <iostream>
7
8  class Vector2D
9  {
10 private:
11
12     float fX;
13     float fY;
14
15 public:
16
17     Vector2D( float aX = 1.0f, float aY = 0.0f ) : fX(aX), fY(aY) {}
18     Vector2D( std::istream& aIStream ) { aIStream >> *this; }
19
20     float getX() const;
21     float getY() const;
22
23     Vector2D operator+( const Vector2D& aVector ) const;
24     Vector2D operator-( const Vector2D& aVector ) const;
25
26     Vector2D operator*( const float aScalar ) const;
27     float dot( const Vector2D& aVector ) const;
28     float cross( const Vector2D& aVector ) const;
29
30     float length() const;
31     Vector2D normalize() const;
32
33     float direction() const;
34     Vector2D align( float aAngleInDegrees ) const;
35
36     friend std::istream& operator>>( std::istream& aIStream, Vector2D& aVector );
37     friend std::ostream& operator<<( std::ostream& aOStream, const Vector2D& aVector );
38 };
39
40 Vector2D operator*( const float aScalar, const Vector2D& aVector );
```

Line: 2:16 C++ Tab Size: 4

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 Composition: Particle2D

Requirements for a particle class

- Particles represent physical model entities with the following attributes:
  - Mass (the particle's resistance - inertia - to change in its velocity)
  - Radius (particles can be rendered as filled circles)
  - Position, a 2D vector to assign a location in 2D space.
  - Velocity, a 2D vector to capture the speed of a particle in a given direction
  - Acceleration, a 2D vector representing the acceleration of a particle in a given direction due to the application of some force (e.g., gravity)
- References
  - Leonard Susskind and George Hrabovsky: The Theoretical Minimum - What You Need to Know to Start Doing Physics. Allen Lane (2013)
  - www.codingmath.com

Particle2D: A Simple 2D Particle Class

Particle2D.h — Introduction

```
#include <iostream>

#include "Vector2D.h"

class Particle2D
{
private:
    float fMass;
    float fRadius;
    Vector2D fPosition;
    Vector2D fVelocity;
    Vector2D fAcceleration;

public:

    explicit Particle2D( float aMass = 0.0f,
                         float aRadius = 10.0f,
                         const Vector2D& aPosition = Vector2D( 0.0f, 0.0f ),
                         const Vector2D& aVelocity = Vector2D( 0.0f, 0.0f ),
                         const Vector2D& aAcceleration = Vector2D( 0.0f, 0.0f ) );

    const float getMass() const;
    const float getRadius() const;
    const Vector2D& getPosition() const;
    const Vector2D& getVelocity() const;
    const Vector2D& getAcceleration() const;

    void applyAcceleration( const Vector2D& aAcceleration );
    float direction() const;
    void align( float aAngleInDegrees );
    float speed() const;

    void update();

    friend std::ostream& operator<<( std::ostream& aOStream, const Particle2D aParticle );
};
```

Line: 2:16 C++ Tab Size: 4

Explicit Particle2D Constructor

Particle2D.h — Introduction

```
#include <iostream>

#include "Vector2D.h"

class Particle2D
{
private:
    float fMass;
    float fRadius;
    Vector2D fPosition;
    Vector2D fVelocity;
    Vector2D fAcceleration;

public:

    explicit Particle2D( float aMass = 0.0f,
                         float aRadius = 10.0f,
                         const Vector2D& aPosition = Vector2D( 0.0f, 0.0f ),
                         const Vector2D& aVelocity = Vector2D( 0.0f, 0.0f ),
                         const Vector2D& aAcceleration = Vector2D( 0.0f, 0.0f ) );

    const float getMass() const;
    const float getRadius() const;
    const Vector2D& getPosition() const;
    const Vector2D& getVelocity() const;
    const Vector2D& getAcceleration() const;

    void applyAcceleration( const Vector2D& aAcceleration );
    float direction() const;
    void align( float aAngleInDegrees );
    float speed() const;

    void update();

    friend std::ostream& operator<<( std::ostream& aOStream, const Particle2D aParticle );
};
```

Line: 2:16 C++ Tab Size: 4

The constructor is marked explicit to prevent automatic type conversion from float to Particle2D when all other parameters are omitted (use default).

Constructor Implementation

Particle2D.cpp — Introduction

```
8  Particle2D::Particle2D( float aMass,
9                          float aRadius,
10                         const Vector2D& aPosition,
11                         const Vector2D& aVelocity,
12                         const Vector2D& aAcceleration ) :
13     fMass(aMass),
14     fPosition(aPosition),
15     fVelocity(aVelocity),
16     fAcceleration(aAcceleration)
17 {}
18
```

Line: 22:2 C++ Size: 4 Particle2D::getMass

We use a constructor initializer list to prevent a default initialization of the Vector2D member variables. Instead, we employ the compiler-generated copy constructor to initialize the Vector2D member variables. (The Qt framework uses this approach extensively.)

The Particle2D Operations

Particle2D.cpp — Introduction

```
43
44  void Particle2D::applyAcceleration( const Vector2D& aAcceleration )
45  {
46      fAcceleration = fAcceleration + aAcceleration;
47  }
48
49  float Particle2D::direction() const
50  {
51      return fVelocity.direction();
52  }
53
54  void Particle2D::align( float aAngleInDegrees )
55  {
56      fVelocity = fVelocity.align( aAngleInDegrees );
57  }
58
59  float Particle2D::speed() const
60  {
61      return fVelocity.length();
62  }
63
64  void Particle2D::update()
65  {
66      fVelocity = fVelocity + fAcceleration;
67      fPosition = fPosition + fVelocity;
68  }
```

Line: 22:2 C++ Tab Size: 4 Particle2D::getMass

Adjust the direction of the velocity

Update velocity and position

The Simulation: Main

Main.cpp — Introduction

```
4   #include <iostream>
5
6   #include "Particle2D.h"
7
8   using namespace std;
9
10  int main()
11  {
12      cout << "A simple particle simulation\n" << endl;
13
14      Particle2D obj( 0.0f,
15                      10.0f,
16                      Vector2D( 10.0f, 20.0f ),
17                      Vector2D( 4.0f, 15.0f ),
18                      Vector2D( 0.0f, -0.1f )
19                    );
20
21      do
22      {
23          cout << obj << endl;
24
25          obj.update();
26      } while ( obj.getPosition().getY() >= 20.0f );
27
28      cout << obj << endl;
29
30      return 0;
31  }
```

Line: 1 C++ Tab Size: 4

Microsoft Visual Studio Debug C...

```
Speed: 14.95
Direction: -74.48
Position: [1186,93]

Speed: 15.04
Direction: -74.58
Position: [1190,79]

Speed: 15.14
Direction: -74.68
Position: [1194,64]

Speed: 15.23
Direction: -74.78
Position: [1198,50]

Speed: 15.33
Direction: -74.88
Position: [1202,35]

Speed: 15.43
Direction: -74.97
Position: [1206,20]
```

Loop until particle is less than 20 units above the base line.

Simulation Visualization

Best Angle: 45 Degrees

Main.cpp — Introduction

```cpp
#include <iostream>

#include "Particle2D.h"

using namespace std;

int main()
{
    cout << "A simple particle simulation\n" << endl;

    Particle2D obj( 0.0f,
                    10.0f,
                    Vector2D( 10.0f, 20.0f ),
                    Vector2D( 4.0f, 15.0f ),
                    Vector2D( 0.0f, -0.1f )
                  );

    obj.align( 45.0f );   // best angle

    do
    {
        cout << obj << endl;

        obj.update();
    } while ( obj.getPosition().getY() >= 20.0f );

    cout << obj << endl;

    return 0;
}
```

Line: 1 C++ Tab Size: 4

Microsoft Visual Studio Debug Cons...

```
Speed: 15.21
Direction: -43.8
Position: [2369,57]

Speed: 15.28
Direction: -44.08
Position: [2380,47]

Speed: 15.35
Direction: -44.34
Position: [2391,36]

Speed: 15.42
Direction: -44.61
Position: [2402,25]

Speed: 15.49
Direction: -44.87
Position: [2413,14]
```

Object Models & Inheritance

C++ Object Models

- C++ supports:
  - A value-based object model
  - A reference-based object model

This can make programming in C++ difficult. Java, for examples, has only one model - everything is a reference!

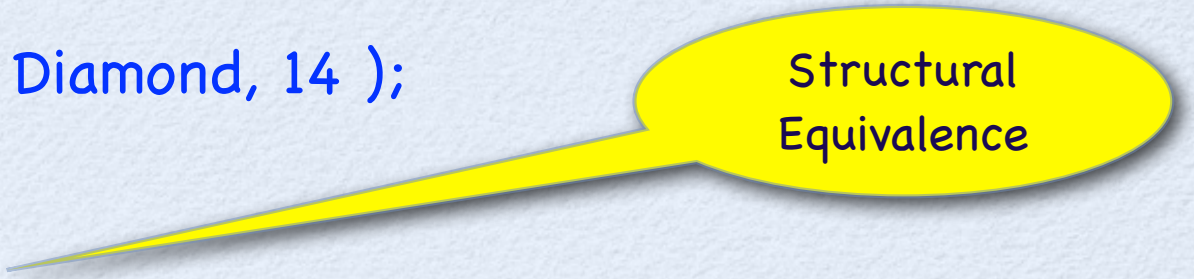
The Value-based Object Model

- Value-based object model:
  - Objects are stored on the stack.
  - Object are accessed through object variables.
  - An object's memory is implicitly released.

Valued-based Objects

- Value-based objects look and feel like records (or structs):

```
Card AceOfDiamond( Diamond, 14 );

Card TestCard( Diamond, 14 );
```




```
if ( TestCard == AceOfDiamond )

    cout << "The test card is " << TestCard.getName() << endl;
```

The Reference-based Object Model

- Reference-based object model:
  - Objects are stored on the heap.
  - Objects are accessed through pointer variables.
  - An object's memory must be explicitly released.

Reference-based Objects

- Reference-based objects require pointer variables and an explicit new and delete:

```
Card* AceOfDiamond = new Card( Diamond, 14 );

Card* TestCard = new Card( Diamond, 14 );


if ( TestCard == AceOfDiamond )

    cout << "The test card is " << TestCard->getName() << endl;


delete AceOfDiamond;

delete TestCard;
```

Cardinal Rule

- We find both models in C++ code.
- Value semantics provides us with objects that behave like arithmetic types:

Every object is unique, but it may be known under numerous aliases (aka references).

Inheritance

- Inheritance lets us define classes that model relationships among classes, sharing what is common, and specializing only that which is inherently different.
- Inheritance is
  - A mechanism for specialization
  - A mechanism for reuse
  - Fundamental to supporting polymorphism

An Account Class

Account.h

```
4   class Account
5   {
6   private:
7       unsigned long long fNumber;
8       double fBalance;
9
10  public:
11      Account( unsigned long long aNumber, double aBalance = 0.0 );
12
13      bool deposit( double aAmount );
14      bool withdraw( double aAmount );
15
16      double getBalance() const;
17  };
```

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

Default argument

- An account has a number (2^{64} - 1 values) and a balance.
- To create an account we need a number. Funds can be credited to an account once it has been created.

Account Implementation

Account.cpp

```
4   Account::Account( unsigned long long aNumber, double aBalance )
5   {
6       fNumber = aNumber;
7       fBalance = aBalance;
8   }
9
10  bool Account::deposit( double aAmount )
11  {
12      fBalance += aAmount;
13
14      return true;
15  }
16
17  bool Account::withdraw( double aAmount )
18  {
19      fBalance -= aAmount;
20
21      return true;
22  }
23
24  double Account::getBalance() const
25  {
26      return fBalance;
27  }
28
```

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

Default in class specification

Signal success

Balance can become negative

A BankAccount Class

BankAccount.h

```
6   class BankAccount : public Account
7   {
8   private:
9       double fInterestRate;
10
11  public:
12      BankAccount( unsigned long long aNumber, double aBalance = 0.0, double aInterestRate = 0.0 );
13
14      void setInterestRate( double aInterestRate );
15      double getInterestRate() const;
16
17      bool creditInterest();
18      bool applyServiceFee( double aFee );
19
20      // inherited from Account:
21      //  bool deposit( double aAmount );
22      //  bool withdraw( double aAmount );
23      //  double getBalance() const;
24  };
```

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

- A bank account is a subtype of account. A bank account includes all features of an account and offers bank account specific additional features (incremental refinement).

Access Levels for Inheritance

- public:
  - Public members in the base class remain public.
  - Protected members in the base class remain protected.
  - Yields a "is a" relationship, that is, a subtype.
- protected:
  - Public and protected members in the base class are protected in the derived class.
  - Yields a "implemented in terms of" relationship, that is, a new type.
- private:
  - Public and protected members in the base class become private in the derived class.
  - Yields a stricter "implemented in terms of" relationship, that is, a new type.

Public Inheritance

- Public inheritance enables inclusion polymorphism: A subclass is a subtype.
- The subtype relationship is a key ingredient in contemporary object-oriented software development.
- An object of a subclass can be used safely anywhere an object of a superclass is expected. Example, we can use a BankAccount in lieu of an Account object.

A Grain of Salt

- Public inheritance is not without flaws. It can result in an unwieldy and confusingly large set of public methods.
- Public inheritance is based on additive refinement. New classes are defined by extending an old class with new features.
- Protected and private inheritance allow for a more fine-grained control of the set of public methods. Unfortunately, these mechanisms do not yield subclasses and is often viewed controversial.

Constructors and Inheritance

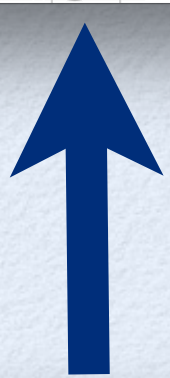
- Whenever an object of a derived class is instantiated, multiple constructors are called so that each class in the inheritance chain can initialize itself.
- The constructor for each class in the inheritance chain is called beginning with the base class at the top of the inheritance chain and ending with the most recent derived class.

Base Class Initializer

Account.cpp

```
4  Account::Account( unsigned long long aNumber, double aBalance )
5  {
6      fNumber = aNumber;
7      fBalance = aBalance;
8  }
```

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



Direct super class constructor

BankAccount.cpp

```
4  BankAccount::BankAccount( unsigned long long aNumber, double aBalance, double aInterestRate ) : Account( aNumber, aBalance )
5  {
6      fInterestRate = aInterestRate;
7  }
```

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

Facts About Base Class Initializers

- If a base class does not have a default constructor, the derived class must provide a base class initializer for it.
- Base class initializers frequently appear alongside member initializers, which use similar syntax.
- If more than one argument is required by a base class constructor, the arguments are separated by comma.
- Reference members need to be initialized using a member initializer.

Destructors and Inheritance

- Whenever an object of a derived class is destroyed, the destructor for each class in the inheritance chain, if defined, is called.
- The destructor for each class in the inheritance chain is called beginning with the most recent derived class and ending with the base class at the top of the inheritance chain.

Account & BankAccount Destructors

Account.h

```
4   class Account
5   {
6   private:
7       unsigned long long fNumber;
8       double fBalance;
9
10  public:
11      Account( unsigned long long aNumber, double aBalance = 0.0 );
12
13      virtual ~Account() {}
14
15      bool deposit( double aAmount );
16      bool withdraw( double aAmount );
17
18      double getBalance() const;
19  };
```

Line: 13 Column: 1 C++ Tab Size: 4 ~Account

destructor (virtual)

Account & BankAccount Destructors

BankAccount.h

```
6  class BankAccount : public Account
7  {
8  private:
9      double fInterestRate;
10
11 public:
12     BankAccount( unsigned long long aNumber, double aBalance = 0.0, double aInterestRate = 0.0 );
13
14     ~BankAccount() override {}
15
16     void setInterestRate( double aInterestRate );
17     double getInterestRate() const;
18
19     bool creditInterest();
20     bool applyServiceFee( double aFee );
21
22     // inherited from Account:
23     //  bool deposit( double aAmount );
24     //  bool withdraw( double aAmount );
25     //  double getBalance() const;
26 };
```

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

overridden destructor (virtual)

Virtual Member Functions

- To give a member function from a base class new behavior in a derived class, one overrides it.
- To allow a member function in a base class to be overridden, one must declare the member function virtual.
- Note, non-virtual member functions are resolved with respect to the declared type of the object.
- In Java all member functions are virtual.
- Explicitly defining a (virtual) destructor affects which special member functions the compiler synthesizes (C++-11).

The C++11 override Keyword

- With C++11, we have a new keyword, override, to signal overridden virtual methods (including destructors).
- It has the same semantics as in C#, but you do not need to explicitly specify it.
- The virtual specifier can still be used in these cases, but override is the new standard.

Virtual Destructors

- When deleting an object using a base class pointer of reference, it is essential that the destructors for each of the classes in the inheritance chain get a chance to run:

```
BankAccount *BAptr;

Account *Aptr;

BAptr = new BankAccount( 2.25 );

Aptr = BAptr;

...

delete Aptr;
```

Virtual Account Destructor

```
class Account
{
public:
    virtual ~Account() { ... }
};
```

Not declaring a destructor virtual is a common source of memory leaks in C++ code.

Method Overriding

- Method overriding is an object-oriented programming mechanism that allows a subclass to provide a more specific implementation of a method, which is also present in one of its superclasses.
- The implementation in the subclass overrides (replaces) the implementation in the superclass by providing a method that has the same name, parameter signature, and return type as the method in the parent class. We say these methods belong to the same method family.
- Which (overridden) method is selected at runtime is determined by the receiver object used to invoke it. In general, the most recent definition is chosen, if possible.

Method Family

- A member function of a class always belongs to a specific set, called method family. If the elements of this set are virtual, then their invocation is governed by dynamic binding, a technique that makes polymorphism real.

Virtual withdraw Method

Account.h

```
4   class Account
5   {
6   private:
7       unsigned long long fNumber;
8       double fBalance;
9
10  public:
11      Account( unsigned long long aNumber, double aBalance = 0.0 );
12      virtual ~Account() {}
13
14      bool deposit( double aAmount );
15
16      virtual bool withdraw( double aAmount );
17
18      double getBalance() const;
19  };
```

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

Virtual withdraw Method

BankAccount.h

```
6  class BankAccount : public Account
7  {
8  private:
9      double fInterestRate;
10
11 public:
12     BankAccount( unsigned long long aNumber, double aBalance = 0.0, double aInterestRate = 0.0 );
13     ~BankAccount() override {}
14
15     bool withdraw( double aAmount ) override;
16
17     void setInterestRate( double aInterestRate );
18     double getInterestRate() const;
19
20     bool creditInterest();
21     bool applyServiceFee( double aFee );
22
23     // inherited from Account:
24     //  bool deposit( double aAmount );
25     //  bool withdraw( double aAmount );
26     //  double getBalance() const;
27 };
```

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

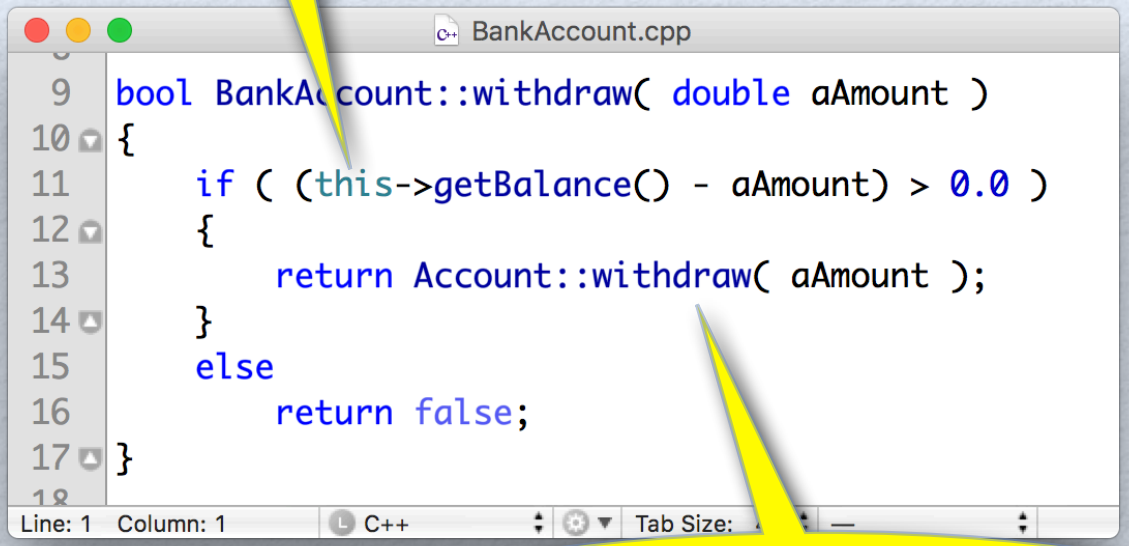
overridden method

The withdraw Specialization

- In class Account, the method withdraw implements the behavior required to take funds out of an account. The Account class does not do any range checks. Hence, the execution of the withdraw method could result in a negative account balance.
- In general, bank accounts do not allow for negative balances (unless it is a credit account). Hence, a withdrawal can only succeed if there are sufficient funds in the account. The BankAccount's withdraw methods must guard a bank account against requests exceeding the current account balance.

Overriding the withdraw Method

Call inherited method (this-> optional)



Call overridden method

Calling a Virtual Method

Main.cpp

```
 8  int main()
 9  {
10      BankAccount lAccount( 12345, 0.0, 0.5 );
11
12      cout << "Balance: " << lAccount.getBalance() << endl;
13
14      // dynamic method invocation
15
16      Account& lBankAccountReference = lAccount;
17
18      if ( lBankAccountReference.withdraw( 50.0 ) )
19      {
20          cout << "We got instant credit. Wow!" << endl;
21      }
22      else
23      {
24          cout << "The bank refused to give us money." << endl;
25      }
26
27      return 0;
28  }
```

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

Calls BankAccount::withdraw

Debug

```
Kamala:Debug Markus$ ./Inheritance
Balance: 0
The bank refused to give us money.
Kamala:Debug Markus$ _
```

Facts About Virtual Members

- Constructors cannot be virtual.
- Declaring a member function virtual does not require that this function must be overridden in derived classes, except the member function is declared pure virtual.
- Once a member function has been declared virtual, it remains virtual.
- Parameter and result types must match to properly override a virtual member function.
- If one declares a non-virtual member function virtual in a derived class, the new member function hides the inherited member function.

Note on Virtual Member Functions

- If a virtual member function is called from inside a constructor or destructor, then the version that is run is the one defined for the type of the constructor or destructor itself - static method invocation.
- C++ is a very complex language.