# FIRST STEPS

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

# THE TOOLS OF THE TRADE: PROGRAMMING LANGUAGES

- Programming languages provide a framework to organize computational complexity in our minds.

- Programming languages offer us effective means by which we communicate our understanding of computerized problem solutions.

# WE CHOOSE C++20.

# WHY C++

- We need to know more than just Java or C#.
- C++ is highly efficient and better suited for implementing low-level software layers, like device controllers or networking protocols.
- C++ is widely used in commercial applications, operating systems, and modern game development.
- In C++, memory is tangible, allowing us to study the effects of design decisions on memory management.
- Additionally, because C++ is close to the hardware, we can analyze performance issues more easily.


# WHAT IS C++

- C++ is a general-purpose, high-level programming language that also includes low-level features.
- In 1983, Bjarne Stroustrup developed C++ at Bell Labs to enhance the C programming language, initially known as "C with Classes."
- The first standard for the C++ programming language ISO/IEC 14882:1998 was ratified in 1998. The most recent version is ISO/IEC 14882:2024, capturing C++23.
- C++11 introduced significant changes compared to C++98. C++14 is a small extension of C++11, while C++17 represents a major revision. C++20 adds even more substantial features than both C++14 and C++17.
- Key features introduced in C++11 include move semantics and lambda expressions (the latter was revised in C++14).
- C++17 simplifies the use of nested namespaces and expands vocabulary types.
- Finally, C++ 20 adds concepts, expands the application of **constexpr**, and eliminates the need for **typename** in certain circumstances.

# DESIGN PHILOSOPHY OF C++

- C++ is a hybrid, statically-typed, general-purpose language known for its efficiency and portability, similar to C.
- C++ directly supports multiple programming styles, including procedural, object-oriented, functional, and generic programming.
- C++ gives programmers the freedom to choose their preferred approach, though it also means programmers can make suboptimal decisions!
- While C++ avoids features that are platform-specific or not general-purpose, it is ultimately platform-dependent.
- The language does not incur overhead for features that are not utilized and operates without a built-in, sophisticated programming environment.

# A SIMPLE I/O PROGRAM

```
#include <iostream>

int main()
{
    std::cout << "Enter two numbers:" << std::endl;

    int v1, v2;

    std::cin >> v1 >> v2;

    std::cout << "The sum of " << v1 << " and " << v2
              << " is " << v1+v2 << std::endl;

    return 0;
}
```

```
Enter two numbers:
5
7
The sum of 5 and 7 is 12
```

- The #include directive is a pre-processor command to add C++ libraries and other project resources to the current compilation unit.
- Standard library abstractions are defined within the C++ namespace std. To refer to elements in this namespace, we used the scope identifier std::.
- The operators << and >> correspond to function names in the C++ standard library. Although not defined in the global namespace, Argument-dependent lookup (ADL) enables us to resolve these operators to their respective function definitions within the std namespace.
- Every C++ program must return an error code to the operating system, where a return value of 0 indicates successful termination of the program.
- By default, formatted input in C++ disregards whitespace characters, which include newlines, spaces, and tabs.

# THE STANDARD INPUT STREAM CIN

- The object cin is an instance of class istream and represents the standard input stream, similar to stdin in C.
- The object cin is globally visible and can be used by any C++ compilation unit (e.g., a .cpp-file) that includes the iostream library.
- The object cin receives input from the keyboard or a stream linked to the standard input.
- We use the extraction operator >> to fetch formatted data or the methods read() or get() to retrieve unformatted data from the standard input stream.

# THE STANDARD OUTPUT STREAM COUT

- The object cout is an instance of class ostream and represents the standard output stream. It corresponds to stdout in C.
- The object cout is globally visible and can be used by any C++ compilation unit (i.e., a .cpp-file) that includes the iostream library.
- The object cout sends data to the console, displaying it as printable characters, or to a stream linked to the standard output.
- We use the operator << to push formatted data, or we can use the methods write() or put() to send unformatted data to the standard output stream.

# C++ PARADIGMS

- C++ is a multi-paradigm language.
- C++ naturally supports both
  - the imperative paradigm and
  - the object-oriented paradigm.
- In any non-trivial project, these paradigms must be mixed.

# IMPERATIVE PROGRAMMING

- Imperative programming is one of the oldest styles of programming. In this style, the algorithm for computation is clearly expressed through a series of instructions, including assignments, tests, and conditional branching.

- To execute an algorithm, data values must be stored in variables that a program can access and modify.

- This programming style aligns naturally with the von Neumann architecture, the foundational model for many computers, and remains widely used today.

# INSERTION SORT (INCREASING ORDER)

```
template<typename T, typename Op = std::greater<T> >
void performInsertionSort( std::vector<T>& aArray, Op aTest = Op{},
                           std::ostream& aOStream = std::cout )
{
    size_t i = 1;

    while ( i < aArray.size() )
    {
        size_t j = i++;

        while ( j > 0 && aTest( aArray[j-1], aArray[j] ) )
        {
            std::swap( aArray[j-1], aArray[j] );

            j--;
        }

        sendToStream( aOStream, aArray, 2u ) << std::endl;
    }
}
```

```
[45,34,8,6,5,1,0,-2,-3,-100]
  [34,45,8,6,5,1,0,-2,-3,-100]
  [8,34,45,6,5,1,0,-2,-3,-100]
  [6,8,34,45,5,1,0,-2,-3,-100]
  [5,6,8,34,45,1,0,-2,-3,-100]
  [1,5,6,8,34,45,0,-2,-3,-100]
  [0,1,5,6,8,34,45,-2,-3,-100]
  [-2,0,1,5,6,8,34,45,-3,-100]
  [-3,-2,0,1,5,6,8,34,45,-100]
  [-100,-3,-2,0,1,5,6,8,34,45]
[-100,-3,-2,0,1,5,6,8,34,45]
```

```
std::vector IArrayA { 45, 34, 8, 6, 5, 1, 0, -2, -3, -100 };

sendToStream( std::cout, IArrayA ) << std::endl;
performInsertionSort( IArrayA );
sendToStream( std::cout, IArrayA ) << std::endl;
```

# OBJECT-ORIENTED PROGRAMMING

- Object-oriented languages are based on the concepts of objects and classes, which can be compared to variables and types in a language like Pascal and C.
- A class describes the characteristics common to all its instances, resembling Pascal records (or structures in C) and thus establishing a set of fields.
- Instead of applying global procedures or functions to variables, we invoke the methods associated with the instances (or objects), a process known as "message passing."
- The core concept of inheritance allows us to create new classes from existing ones by modifying or extending the inherited classes. Inheritance is a fundamental technique that enables code reuse and refinement in object-oriented programming languages.

# INSERTION SORTER CLASS

```
template<typename T>
class InsertionSorter
{
public:

    using array_type = std::vector<T>;

    InsertionSorter( const array_type& aData ) : fData(aData) {}

    template<typename Op = std::greater<T>>
    void operator()( Op aTest = Op{},
                     std::ostream& aOStream = std::cout )
    {
        performInsertionSort( fData, aTest, aOStream );
    }

    const array_type& data() const { return fData; }

private:

    array_type fData;
};
```

```
std::vector IArrayB { 45, 34, 8, 6, 5, 1, 0, -2, -3, -100 };
InsertionSorter ISorter( IArrayB );

sendToStream( std::cout, ISorter.data() ) << std::endl;
ISorter();
sendToStream( std::cout, ISorter.data() ) << std::endl;
sendToStream( std::cout, ISorter.data() ) << std::endl;
ISorter( std::less<int>{} );
sendToStream( std::cout, ISorter.data() ) << std::endl;
```

```
[45,34,8,6,5,1,0,-2,-3,-100]
 [34,45,8,6,5,1,0,-2,-3,-100]
 [8,34,45,6,5,1,0,-2,-3,-100]
 [6,8,34,45,5,1,0,-2,-3,-100]
 [5,6,8,34,45,1,0,-2,-3,-100]
 [1,5,6,8,34,45,0,-2,-3,-100]
 [0,1,5,6,8,34,45,-2,-3,-100]
 [-2,0,1,5,6,8,34,45,-3,-100]
 [-3,-2,0,1,5,6,8,34,45,-100]
 [-100,-3,-2,0,1,5,6,8,34,45]
[-100,-3,-2,0,1,5,6,8,34,45]
[-100,-3,-2,0,1,5,6,8,34,45]
 [-3,-100,-2,0,1,5,6,8,34,45]
 [-2,-3,-100,0,1,5,6,8,34,45]
 [0,-2,-3,-100,1,5,6,8,34,45]
 [1,0,-2,-3,-100,5,6,8,34,45]
 [5,1,0,-2,-3,-100,6,8,34,45]
 [6,5,1,0,-2,-3,-100,8,34,45]
 [8,6,5,1,0,-2,-3,-100,34,45]
 [34,8,6,5,1,0,-2,-3,-100,45]
 [45,34,8,6,5,1,0,-2,-3,-100]
[45,34,8,6,5,1,0,-2,-3,-100]
```

# SENDTOSTREAM

```
template<typename T>
std::ostream& sendToStream( std::ostream& aOStream,
                            const std::vector<T>& aArray,
                            size_t aIndent = 0 )
{
    for ( size_t i = 0; i < aIndent; i++ )
    {
        aOStream << ' ';
    }

    aOStream << '[';

    if ( aArray.size() > 0 )
    {
        aOStream << aArray[0];

        for ( size_t i = 1; i < aArray.size(); i++ )
        {
            aOStream << ',' << aArray[i];
        }
    }

    return aOStream << ']';
}
```

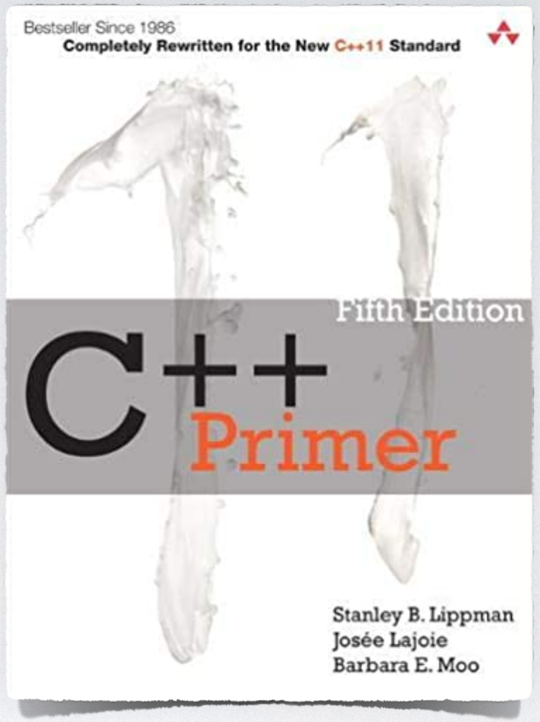
- C++ does not directly define any I/O (input/output) primitives.
- Instead, I/O operations are handled by standard libraries.
- sendToStream() is a generic function that can be applied to any value of type T with a defined formatted output operator.
- We will explore generic abstractions later on.

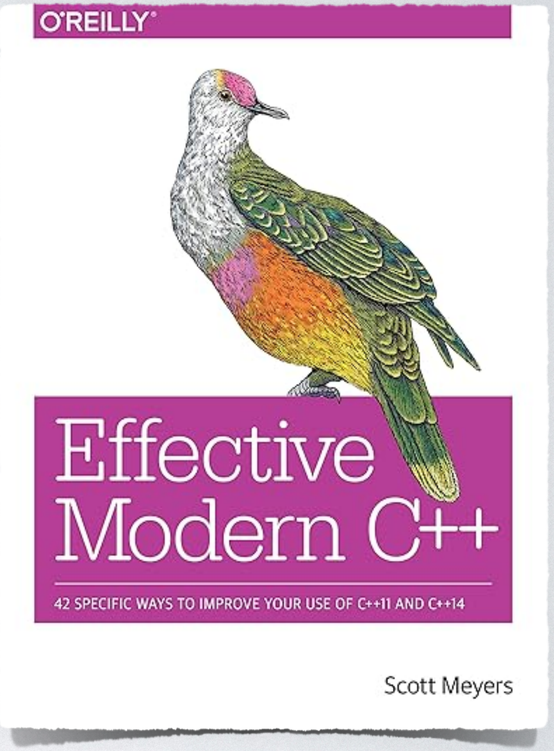
```
std::vector IArrayA { 45, 34, 8, 6, 5, 1, 0, -2, -3, -100 };

sendToStream( std::cout, IArrayA ) << std::endl;
```

```
[45,34,8,6,5,1,0,-2,-3,-100]
```

# A VERY USEFUL LEARNING RESOURCES

- **Stanley B. Lippman, Josée Lajoie, and Barbara E. Moo: C++ Primer. 5th Edition. Addison-Wesley (2013)**:
  - This edition of C++ Primer covers C++11.
  - The 6th Edition may be published February 25, 2025.
- **Scott Meyers: Effective Modern C++. O'Reilly (2014)**:
  - The text aims at the effective application of the features of C++11 and C++14.



# COS30008
# 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 apply to 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 as informal as the habitual coding practices of an individual.
- Compilers do not enforce coding conventions, C++ template meta-programming being the exception.

# MEDIAL CAPITALS

- CamelCase (or medial capitals) is a practice of writing names of variables 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 or acronyms in an identifier should be in caps.
  - Java - variables are mixed-case with a lowercase first letter, methods should be verbs, mixed-case with the first letter lowercase, and classes should be nouns, mixed-case with the first letter of each internal word capitalized.

# BORLAND-INSPIRED STYLE GUIDE ELEMENTS

- Naming conventions:
  - 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'.
- Exception: Loop variables
  - Historically, loop variables are denoted by the letters i, j, and k. This is a convention going back to Fortran and a rule that stated that "every name starting with the letter I, J, K, L, M, N is of type integer".

# INDENTATION STYLE

```
int doFunction( int aParameter1, int aParameter2 )
{
    while (x == y)
    {
        doSomething();
        doSomethingElse();
    }

    doFinalThing();

    return Result;
}
```

- The brace associated with a control statement is put on the next line, indented to the same level as the control statement. Statements within the braces are indented to the next level.
- The indented code is set apart from the containing statement, and the closing brace lines up in the same column as the opening brace.
- This style is less likely to introduce syntax errors via dangling or missing braces.

# WHICH CODING STANDARD TO USE

- Compilers do not enforce coding conventions.
- Not following some or all of the rules may not impact the executable programs created from the source code.
- Code standards facilitate program comprehension and make program maintenance easier.
- Every organization may enforce coding standards as part of its quality assurance process.

# A WARMUP APPLICATION

# A SIMPLE PARTICLE SIMULATION

- Before writing a program in a new language, we must know some of its basic features. C++ is no exception.
- Let us start with a simple particle simulation based on ideas presented by Keith Peters at 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 (physics simulation using semi-implicit Euler method)

# TECHNICAL REFERENCES

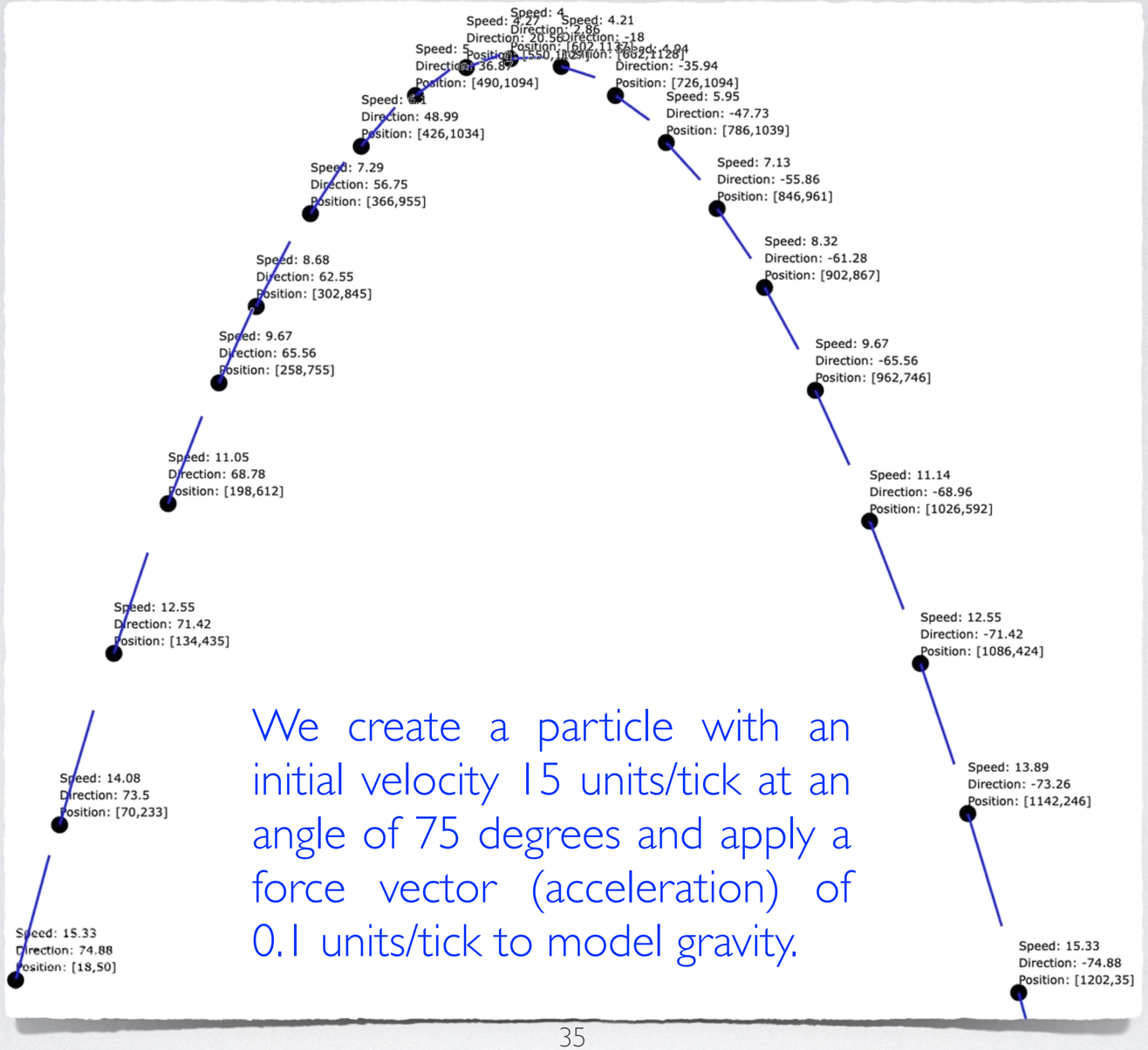

- Stanley B. Lippman, Josée Lajoie, and Barbara E. Moo: **C++ Primer**. 5th Edition. Addison-Wesley (2013)
- Scott Meyers: **Effective Modern C++**. O'Reilly (2014)
- Anthony Williams: **C++ Concurrency in Action - Practical Multithreading**. Manning Publications Co. (2012)
- Eric Lengyel: **Mathematics for 3D Game Programming and Computer Graphics**, Course Technology (2012)
- 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)
- Keith Peters: **www.codingmath.com**

# SOMETHING TO REMEMBER

François Chollet (creator of Keras):

> "The most precise, unambiguous description of a mathematical operation is its executable code."

# 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) of 0.1 units/tick to model gravity.

# A BASIC 2D VECTOR CLASS

```
#pragma once

#include <iostream>

class Vector2D
{
private:
    float fX;
    float fY;

public:
    Vector2D( float aX = 1.0f, float aY = 0.0f ) noexcept;
    Vector2D( std::istream& aIStream ) : Vector2D() { aIStream >> *this; }

    float x() const noexcept { return fX; }
    float y() const noexcept { return fY; }

    Vector2D operator+( const Vector2D& aRHS ) const noexcept;
    Vector2D operator-( const Vector2D& aRHS ) const noexcept;

    Vector2D operator*( const float aRHS ) const noexcept;
    float dot( const Vector2D& aRHS ) const noexcept;
    float cross( const Vector2D& aRHS ) const noexcept;

    float length() const noexcept;
    Vector2D normalize() const noexcept;

    float direction() const noexcept;
    Vector2D align( float aAngleInDegrees ) const noexcept;

    friend std::istream& operator>>( std::istream& aIStream, Vector2D& aVector );
    friend std::ostream& operator<<( std::ostream& aOStream, const Vector2D& aVector );
};

Vector2D operator*( const float aScalar, const Vector2D& aVector ) noexcept;
```

- Vectors are objects that live in multi-dimensional spaces and have their own notion of arithmetic.
- Class Vector2D models two-dimensional (2D) vectors, easy to visualize and compute.
- 2D vectors allow us to develop a mental model when reasoning about higher-dimensional problems.
- Vector2D defines a trivially copyable type whose storage is contiguous (the associated memory block can be copied via memcpy). A trivially copyable type/class has no virtual members and does not explicitly define a destructor, copy, and move operations.

# STRUCTURE OF A CLASS

```
class X
{
private:

    // private members

protected:

    // protected members

public:

    // public members
};
```

Don't forget the semicolon.

- Access levels can be arbitrarily interleaved. It is possible to define multiple sections with the same visibility modifier. The compiler synthesizes one for each, if present.

# ACCESS LEVEL 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.
- **Note**:

  Within a method of an object, you can access members of another object irrespective of their defined access level if the type/class of the other object is the same as **this** object.

# VECTOR2D: PRIVATE MEMBERS

Vector coordinates

```
#pragma once

#include <iostream>

class Vector2D
{
private:
    float fX;
    float fY;

public:
    Vector2D( float aX = 1.0f, float aY = 0.0f ) noexcept;
    Vector2D( std::istream& aIStream ) : Vector2D() { aIStream >> *this; }

    float x() const noexcept { return fX; }
    float y() const noexcept { return fY; }

    Vector2D operator+( const Vector2D& aRHS ) const noexcept;
    Vector2D operator-( const Vector2D& aRHS ) const noexcept;

    Vector2D operator*( const float aRHS ) const noexcept;
    float dot( const Vector2D& aRHS ) const noexcept;
    float cross( const Vector2D& aRHS ) const noexcept;

    float length() const noexcept;
    Vector2D normalize() const noexcept;

    float direction() const noexcept;
    Vector2D align( float aAngleInDegrees ) const noexcept;

    friend std::istream& operator>>( std::istream& aIStream, Vector2D& aVector );
    friend std::ostream& operator<<( std::ostream& aOStream, const Vector2D& aVector );
};

Vector2D operator*( const float aScalar, const Vector2D& aVector ) noexcept;
```

# VECTOR2D: PRIVATE MEMBERS

```
#pragma once

#include <iostream>

class Vector2D
{
private:
    float fX;
    float fY;

public:
    Vector2D( float aX = 1.0f, float aY = 0.0f ) noexcept;
    Vector2D( std::istream& aIStream ) : Vector2D() { aIStream >> *this; }

    float x() const noexcept { return fX; }
    float y() const noexcept { return fY; }

    Vector2D operator+( const Vector2D& aRHS ) const noexcept;
    Vector2D operator-( const Vector2D& aRHS ) const noexcept;

    Vector2D operator*( const float aRHS ) const noexcept;
    float dot( const Vector2D& aRHS ) const noexcept;
    float cross( const Vector2D& aRHS ) const noexcept;

    float length() const noexcept;
    Vector2D normalize() const noexcept;

    float direction() const noexcept;
    Vector2D align( float aAngleInDegrees ) const noexcept;

    friend std::istream& operator>>( std::istream& aIStream, Vector2D& aVector );
    friend std::ostream& operator<<( std::ostream& aOStream, const Vector2D& aVector );
};

Vector2D operator*( const float aScalar, const Vector2D& aVector ) noexcept;
```

Basic vector operations

# VECTOR2D: FRIENDS

```
#pragma once

#include <iostream>

class Vector2D
{
private:
    float fX;
    float fY;

public:
    Vector2D( float aX = 1.0f, float aY = 0.0f ) noexcept;
    Vector2D( std::istream& aIStream ) : Vector2D() { aIStream >> *this; }

    float x() const noexcept { return fX; }
    float y() const noexcept { return fY; }

    Vector2D operator+( const Vector2D& aRHS ) const noexcept;
    Vector2D operator-( const Vector2D& aRHS ) const noexcept;

    Vector2D operator*( const float aRHS ) const noexcept;
    float dot( const Vector2D& aRHS ) const noexcept;
    float cross( const Vector2D& aRHS ) const noexcept;

    float length() const noexcept;
    Vector2D normalize() const noexcept;

    float direction() const noexcept;
    Vector2D align( float aAngleInDegrees ) const noexcept;

    friend std::istream& operator>>( std::istream& aIStream, Vector2D& aVector );
    friend std::ostream& operator<<( std::ostream& aOStream, const Vector2D& aVector );
};

Vector2D operator*( const float aScalar, const Vector2D& aVector ) noexcept;
```

Support for C++ stream-based I/O

# VECTOR2D: AD HOC DEFINITIONS

```
#pragma once

#include <iostream>

class Vector2D
{
private:
    float fX;
    float fY;

public:
    Vector2D( float aX = 1.0f, float aY = 0.0f ) noexcept;
    Vector2D( std::istream& aIStream ) : Vector2D() { aIStream >> *this; }

    float x() const noexcept { return fX; }
    float y() const noexcept { return fY; }

    Vector2D operator+( const Vector2D& aRHS ) const noexcept;
    Vector2D operator-( const Vector2D& aRHS ) const noexcept;

    Vector2D operator*( const float aRHS ) const noexcept;
    float dot( const Vector2D& aRHS ) const noexcept;
    float cross( const Vector2D& aRHS ) const noexcept;

    float length() const noexcept;
    Vector2D normalize() const noexcept;

    float direction() const noexcept;
    Vector2D align( float aAngleInDegrees ) const noexcept;

    friend std::istream& operator>>( std::istream& aIStream, Vector2D& aVector );
    friend std::ostream& operator<<( std::ostream& aOStream, const Vector2D& aVector );
};

Vector2D operator*( const float aScalar, const Vector2D& aVector ) noexcept;
```

S * V

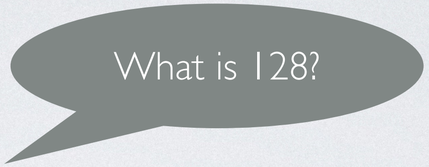
# NOEXCEPT, CONSTANTS, VALUES, AND REFERENCES

# NOEXCEPT

- The C++ Core Guidelines: https://github.com/isocpp/CppCoreGuidelines
- F.6: If your function must not throw, declare it **noexcept**.
  - "If an exception is not supposed to be thrown, the program cannot be assumed to cope with the error and should be terminated as soon as possible. Declaring a function noexcept helps optimizers by reducing the number of alternative execution paths. It also speeds up the exit after failure."
  - Default constructors, destructors, move operations, and swap functions should never throw.
  - Don't apply **noexcept** everywhere. Consider if the possible exceptions can be handled.

# CONSTANTS AND CONST QUALIFIER

- What are the problems with




```
for ( size_t index = 0; index < 128; index++ ) { … }
```


- We can do better

```
for ( size_t index = 0; index < BufferSize; index++ ) { … }
```

- Defining const objects:

```
const int BufferSizeR = getBufferSize();      // initialized at runtime time

constexpr int BufferSizeC = 128;              // initialized at compile time
```

- Unlike macro definitions, const objects have an address!

- All **constexpr** variables are **const**.

# CONSTANT MEMBER FUNCTIONS

- Const member functions prevent accidental changes to the receiver object (i.e., **this** object).
- Const member functions have read-only access to the member variables of the receiver object.
- A member function can be made constant by appending the keyword **const** to the function prototype.
- Objects whose member functions are all marked constant are immutable.
- Note: Non-member functions cannot be declared constant.

# VECTOR2D: CONST MEMBER FUNCTIONS

```
#pragma once

#include <iostream>

class Vector2D
{
private:
    float fX;
    float fY;

public:
    Vector2D( float aX = 1.0f, float aY = 0.0f ) noexcept;
    Vector2D( std::istream& aIStream ) : Vector2D() { aIStream >> *this; }

    float x() const noexcept { return fX; }
    float y() const noexcept { return fY; }

    Vector2D operator+( const Vector2D& aRHS ) const noexcept;
    Vector2D operator-( const Vector2D& aRHS ) const noexcept;

    Vector2D operator*( const float aRHS ) const noexcept;
    float dot( const Vector2D& aRHS ) const noexcept;
    float cross( const Vector2D& aRHS ) const noexcept;

    float length() const noexcept;
    Vector2D normalize() const noexcept;

    float direction() const noexcept;
    Vector2D align( float aAngleInDegrees ) const noexcept;

    friend std::istream& operator>>( std::istream& aIStream, Vector2D& aVector );
    friend std::ostream& operator<<( std::ostream& aOStream, const Vector2D& aVector );
};

Vector2D operator*( const float aScalar, const Vector2D& aVector ) noexcept;
```

EVERYTHING THAT SHOULD OR MUST NOT CHANGE IS MARKED WITH THE CONST KEYWORD.

# LVALUES AND RVALUES

- Expressions in C++ are characterized by two independent properties: a *type* and a *value category*. Two major value categories are lvalue and rvalue. Historically, these value categories correspond to occurrences of expressions on the left and right-hand side of an assignment statement. In basic terms, the lvalue of a variable is the location in memory, and the rvalue is the contents of this location.

- An lvalue is an expression that yields an object reference, such as a variable name, an array subscript reference, a dereferenced pointer, or a function class that returns a reference. An lvalue has a defined region of storage, and consequently, we can obtain an lvalue's address.

- An rvalue is an expression that is not an lvalue. Literals, results of most operators, and function calls that return non-references are rvalues (the compiler uses temporaries for results). An rvalue does not normally have a storage associated with it, therefore, we can't obtain an address of an rvalue. Rvalues can only appear on the right side of an assignment expression.

# LVALUE VS RVALUE

```
#define rvalue 42

int lvalue;

lvalue = rvalue;
```

- The variable lvalue has a defined region of storage in memory.
- We can take its address (e.g., &lvalue).
- The value of rvalue is stored at the location denoted by the address of lvalue.

# LVALUE REFERENCES

BufferSize is an alias to BlockSize

```
int BlockSize = 512;

int& BufferSize = BlockSize;
const int& FixedBufferSize = BlockSize;
```

FixedBufferSize is an read-only alias to BlockSize

- We can think of an lvalue reference as another name for an object.
- Any object whose address can be converted into a given pointer can also be converted into a similar reference type.
- Lvalue references are often more efficient when passing large objects to functions.
- Don't confuse reference declarations with the use of the address-of operator. Both entail the use of the **&** symbol. When &identifier **is preceded** by a type, identifier is declaration as a **reference to the type**. When &identifier **is not preceded** by a type, we mean the **address of the identifier**.

# PARAMETER PASSING MECHANISMS

- C++ uses call-by-value as the 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

# CODE ORGANIZATION

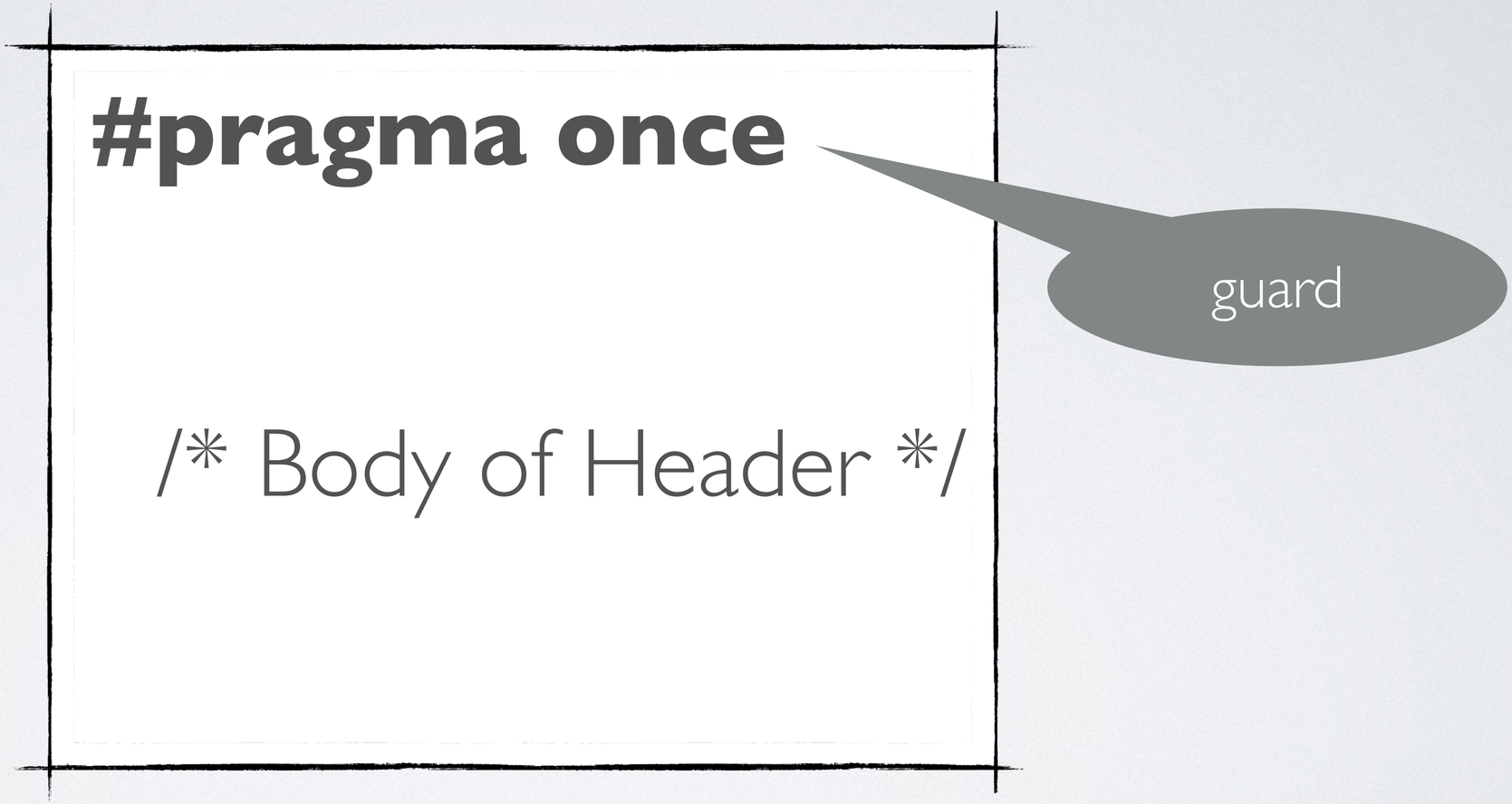
- Specifications go into header files (i.e., *.h or *.hpp).
- Definitions (aka code) go into source files (i.e., *.cpp)
- It is possible to define code in the header files (e.g. x() and y()).
- There is no "The Way" for organizing code. Unless you define a template class, definitions should go into source files. Small functions might be defined in header files. This allows for inlining but comes at the expense of compile time.
- Note: For templates to be instantiated, the compiler must have access to all code. Hence, all code must be defined in the header. Template classes have no source files.

# HEADER FILE: VECTOR2D.H

```
#pragma once

#include <iostream>

class Vector2D
{

        ...


};
```

Implementation goes to Vector2D.cpp

# PROTECTION AGAINST REPEAT INCLUSION



# PROTECTION AGAINST REPEAT INCLUSION - OLD STYLE

guard

```
#ifndef HEADER_H_
#define HEADER_H_


  /* Body of Header */


#endif /* HEADER_H_ */
```

# SOURCE FILE: VECTOR2D.CPP

```
#include "Vector2D.h"

#include <cmath>
#include <cassert>

Vector2D Vector2D::operator+( const Vector2D& aOther ) const
{
    return Vector2D( x() + aOther.x(), y() + aOther.y() );
}
```

Scope identifier as method name

# OBJECT INITIALIZATION

- A class defines a new data type. Instances of this data type are objects that need initialization.
- Each class explicitly (or implicitly) defines some special member functions, called constructors, that are executed whenever we create new objects of a class.
- The constructor ensures that the data members of each object are set to 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 does not take any arguments.
- The compiler will synthesize a default constructor when no other constructors have been specified.
- When the compiler needs to create objects in an environment-agnostic way (e.g., array of objects), it relies on the presence of the default constructor.
- If some data members have built-in or compound types, the class should not rely on the synthesized default constructor!

# VECTOR2D: CONSTRUCTORS

```
#pragma once

#include <iostream>

class Vector2D
{
private:
    float fX;
    float fY;

public:
    Vector2D( float aX = 1.0f, float aY = 0.0f ) noexcept;
    Vector2D( std::istream& aIStream ) : Vector2D() { aIStream >> *this; }

    float x() const noexcept { return fX; }
    float y() const noexcept { return fY; }

    Vector2D operator+( const Vector2D& aRHS ) const noexcept;
    Vector2D operator-( const Vector2D& aRHS ) const noexcept;

    Vector2D operator*( const float aRHS ) const noexcept;
    float dot( const Vector2D& aRHS ) const noexcept;
    float cross( const Vector2D& aRHS ) const noexcept;

    float length() const noexcept;
    Vector2D normalize() const noexcept;

    float direction() const noexcept;
    Vector2D align( float aAngleInDegrees ) const noexcept;

    friend std::istream& operator>>( std::istream& aIStream, Vector2D& aVector );
    friend std::ostream& operator<<( std::ostream& aOStream, const Vector2D& aVector );
};

Vector2D operator*( const float aScalar, const Vector2D& aVector ) noexcept;
```

Default arguments render this constructor default

# CONSTRUCTOR INITIALIZER

```
Vector2D::Vector2D( float aX, float aY ) noexcept :
    fX(aX),
    fY(aY)
{}
```

- A constructor initializer is a comma-separated list of member initializers, declared between the signature and its body.
- Constructor initializers take the form of function calls coinciding with the name of the instance variable being initialized.
- Constructor initializers take precedence over any other (default) initialization.

# IMPLICIT TYPE CONVERSION

- A constructor that can take a single argument may define an implicit conversion from the parameter to the class type.

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

Conversion from std::istream to Vector2D

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

```
Vector2D vecA( 1.0, 2.0 );
Vector2D vecB = vecA + std::cin;
```

# SUPPRESSING IMPLICIT TYPE CONVERSION

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

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

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

```
Vector2D vecA( 1.0, 2.0 );
Vector2D vecB = vecA + std::cin;
```

**Error**

# IT IS APPLICATION-SPECIFIC WHETHER IMPLICIT TYPE CONVERSION NEEDS TO BE SUPPORTED OR NOT.

# OPERATOR OVERLOADING

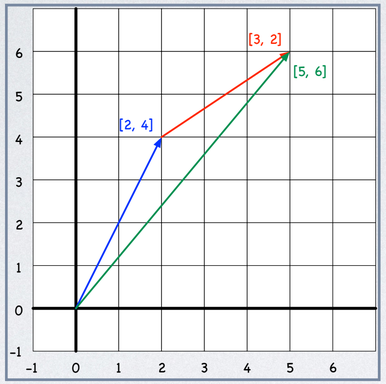
- C++ supports operator overloading.
- Overloaded operators are like normal functions but are defined using a pre-defined operator symbol.
- You cannot change the priority and associativity of an operator.
- Operators are selected by the compiler based on the static types of the specified operands.
- Using operator overloading, we can naturally express a data type-specific notion of arithmetic (adding, multiplying, and so on).

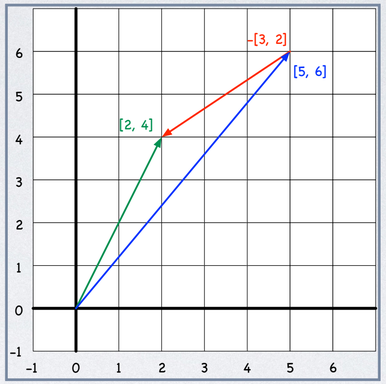
# MEMBER OPERATORS VS AD HOC OPERATORS

- There are two forms of operator overloading:
  - member operator
  - ad hoc operator
- A member operator receives a first implicit argument - **this** - a pointer to the receiver object. In other words, an overloaded operator in C++ is like any other non-static method function defined for a class. Consequently, 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 being defined. An ad hoc definition of a binary operator '+' requires two arguments: the left-hand and the right-hand side of '+'.

# ADDITION & SUBTRACTION

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




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

```
Vector2D Vector2D::operator+( const Vector2D& aRHS ) const noexcept
{
    return Vector2D( x() + aRHS.x(), y() + aRHS.y() );
}

Vector2D Vector2D::operator-( const Vector2D& aRHS ) const noexcept
{
    return Vector2D( x() - aRHS.x(), y() - aRHS.y() );
}
```

# SCALAR MULTIPLICATION, DOT PRODUCT, AND CROSS PRODUCT

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

```
Vector2D Vector2D::operator*( const float aRHS ) const noexcept
{
    return Vector2D( x() * aRHS, y() * aRHS );
}

float Vector2D::dot( const Vector2D& aRHS ) const noexcept
{
    return x() * aRHS.x() + y() * aRHS.y();
}

float Vector2D::cross( const Vector2D& aRHS ) const noexcept
{
    return x() * aRHS.y() - y() * aRHS.x();
}
```

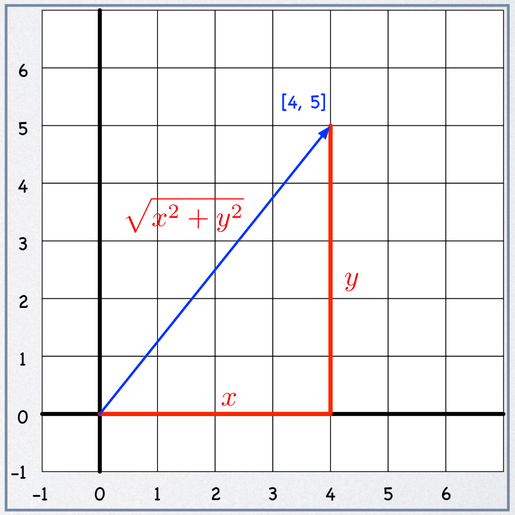
# VECTOR LENGTH AND UNIT VECTOR OF A VECTOR

- 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.)

```
float Vector2D::length() const noexcept
{
    float val = std::sqrt( x() * x() + y() * y() );

    return std::round( val * 100.0f ) / 100.0f;
}

Vector2D Vector2D::normalize() const noexcept
{
    assert( length() != 0.0f );

    return *this * (1.0f/length());
}
```

vector length = Pythagorean theorem



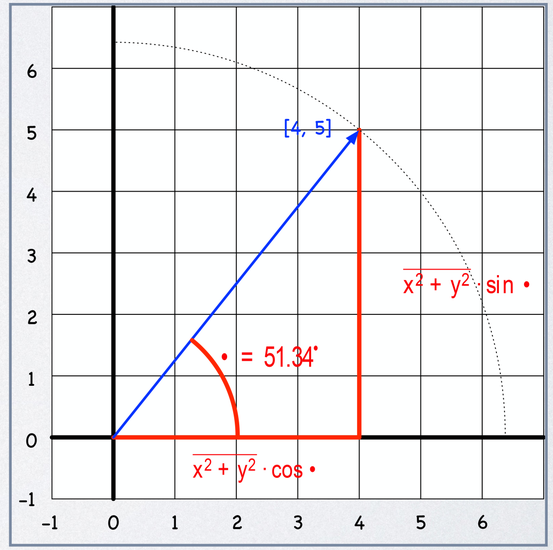
# DIRECTION AND ALIGN

- 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.

```
float Vector2D::direction() const noexcept
{
    float val = std::atan2( y(), x() ) * 180.0f / pif;

    return std::round( val * 100.0f ) / 100.0f;
}

Vector2D Vector2D::align( float aAngleInDegrees ) const noexcept
{
    float lRadians = aAngleInDegrees * pif / 180.0f;

    return length() * Vector2D( std::cos( lRadians ), std::sin( lRadians ) );
}
```

direction/align = basic trigonometry



# DEFINING PI

- Up to C++20, no standard introduced a constant that would represent the number pi ($\pi$).
- We can approximate pi by, say, 3.14159265358979323846 and write:

```
constexpr double pi = 3.14159265358979323846;
```

- It works but can cause problems when working with float and long double expressions.
- Using C++20 concepts, the number pi might be defined as follows (the variable pif represents the number pi in **float**):

```
template<std::floating_point T>
constexpr T pi_v = T(3.14159265358979323846);

constexpr float pif = pi_v<float>;
```

# AD HOC OPERATOR *

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

```
Vector2D operator*( const float aScalar, const Vector2D& aVector ) noexcept
{
    return aVector * aScalar;
}
```

# DATA I/O

# THE VECTOR2D INPUT OPERATOR >>

- The input operator performs formatted input of the vector coordinates; 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 0.0f.
- The input operator returns the stream object, permitting a "chaining" of input operations.

```
std::istream& operator>>( std::istream& aIStream, Vector2D& aVector )
{
    return aIStream >> aVector.fX >> aVector.fY;
}
```

# THE VECTOR2D OUTPUT OPERATOR <<

- The output operator assigns a "textual representation" to objects of type Vector2D.
- It performs formatted output and renders the coordinate values as integers. The round function yields the nearest integral value.
- The input operator returns the stream object, permitting a "chaining" of output operations.

```
std::ostream& operator<<( std::ostream& aOStream, const Vector2D& aVector )
{
    return aOStream << "[" << round( aVector.fX ) << "," << round( aVector.fY ) << "]";
}
```

# 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.

# VECTOR2D: FRIENDS

```
#pragma once

#include <iostream>

class Vector2D
{
private:
    float fX;
    float fY;

public:
    Vector2D( float aX = 1.0f, float aY = 0.0f ) noexcept;
    Vector2D( std::istream& aIStream ) : Vector2D() { aIStream >> *this; }

    float x() const noexcept { return fX; }
    float y() const noexcept { return fY; }

    Vector2D operator+( const Vector2D& aRHS ) const noexcept;
    Vector2D operator-( const Vector2D& aRHS ) const noexcept;

    Vector2D operator*( const float aRHS ) const noexcept;
    float dot( const Vector2D& aRHS ) const noexcept;
    float cross( const Vector2D& aRHS ) const noexcept;

    float length() const noexcept;
    Vector2D normalize() const noexcept;

    float direction() const noexcept;
    Vector2D align( float aAngleInDegrees ) const noexcept;

    friend std::istream& operator>>( std::istream& aIStream, Vector2D& aVector );
    friend std::ostream& operator<<( std::ostream& aOStream, const Vector2D& aVector );
};

Vector2D operator*( const float aScalar, const Vector2D& aVector ) noexcept;
```

Support for C++ stream-based I/O

# THE FRIEND MECHANISMS

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

```
#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 ) ) noexcept;

    float mass() const noexcept { return fMass; }
    float radius() const noexcept { return fRadius; }
    const Vector2D& position() const noexcept { return fPosition; }
    const Vector2D& velocity() const noexcept { return fVelocity; }
    const Vector2D& acceleration() const noexcept { return fAcceleration; }

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

    void update() noexcept;

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

Particle2D objects are trivially copyable.

# EXPLICIT PARTICLE2D CONSTRUCTOR

```
#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 ) ) noexcept;

    float mass() const noexcept { return fMass; }
    float radius() const noexcept { return fRadius; }
    const Vector2D& position() const noexcept { return fPosition; }
    const Vector2D& velocity() const noexcept { return fVelocity; }
    const Vector2D& acceleration() const noexcept { return fAcceleration; }

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

    void update() noexcept;

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

The constructor is marked explicit to prevent automatic type conversion from float to Particle2D when all but the first parameter is omitted.

# CONSTRUCTOR IMPLEMENTATION

```
Particle2D::Particle2D( float aMass,
                        float aRadius,
                        const Vector2D& aPosition,
                        const Vector2D& aVelocity,
                        const Vector2D& aAcceleration ) noexcept :
    fMass(aMass),
    fRadius(aRadius),
    fPosition(aPosition),
    fVelocity(aVelocity),
    fAcceleration(aAcceleration)
{}
```

We use a constructor initializer list to prevent default initialization and employ the compiler-generated copy constructor to initialize the Vector2D member variables.

# PARTICLE2D OPERATIONS

```
void Particle2D::applyAcceleration( const Vector2D& aAcceleration ) noexcept
{
    fAcceleration = fAcceleration + aAcceleration;
}

float Particle2D::direction() const noexcept
{
    return fVelocity.direction();
}

void Particle2D::align( float aAngleInDegrees ) noexcept
{
    fVelocity = fVelocity.align( aAngleInDegrees );
}

float Particle2D::speed() const noexcept
{
    return fVelocity.length();
}

void Particle2D::update() noexcept
{
    fVelocity = fVelocity + fAcceleration;
    fPosition = fPosition + fVelocity;
}
```

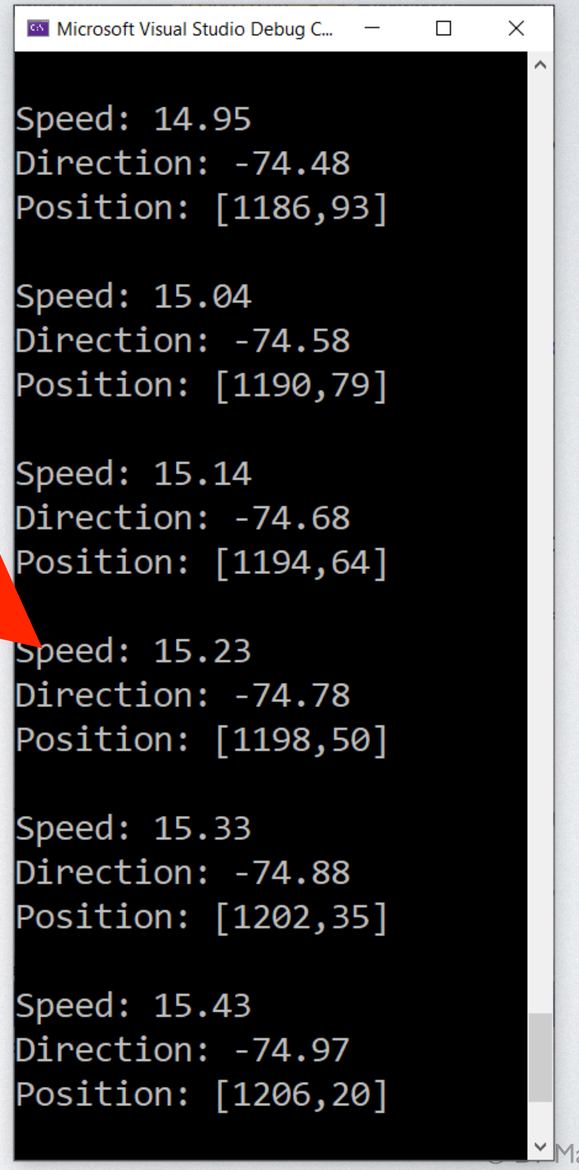
adjust direction of velocity

update velocity and position

# THE SIMULATION: MAIN

```
#include <iostream>

#include "Particle2D.h"

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

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

    do
    {
        std::cout << obj << std::endl;

        obj.update();
    } while ( obj.position().y() >= 20.0f );

    std::cout << obj << std::endl;

    return 0;
}
```

Loop until the particle is less than 20 units above the baseline.



# SIMULATION VISUALIZATION

# BEST ANGLE: 45 DEGREES

```
#include <iostream>

#include "Particle2D.h"

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

    do
    {
        std::cout << obj << std::endl;

        obj.update();
    } while ( obj.position().y() >= 20.0f );

    std::cout << obj << std::endl;

    return 0;
}
```