

# FINM 326: Computing for Finance in C++

## Lecture 1

Chanaka Liyanarachchi

January 6, 2023

Course Info

Getting Started

The C++ Standard Library: A First Look

Data Types and Operators

Functions

Appendix

- Appendix A: Getting Started with Visual Studio

- Appendix B: Getting Started with CLion

- Appendix C: Binary Representation

## Course Info

► Instructor:  
Chanaka Liyanarachchi (chanaka@uchicago.edu)

► Teaching Assistants:

1. Johnny Roches (jaroches@gmail.com)
2. Alex Kay (akay@uchicago.edu)
3. Viren Desai (vdd@uchicago.edu)

# What We Cover

- ▶ This course is about using C++ to solve computing problems in Finance.
- ▶ We learn:
  1. C++ and related programming concepts.
  2. Writing *good quality* code using:
    - ▶ appropriate features and techniques
    - ▶ good programming practices
  3. Use them to write applications in finance.
- ▶ Target audience:
  1. Group 1: No prior C++/programming experience.
  2. Group 2: Have some C++ experience, but you want to learn more and improve.
- ▶ Tentative course schedule posted. Take a look to make sure this is a suitable course for you.
  - ▶ We won't cover everything about C++ in this course.
  - ▶ This course will not make you a "professional C++ developer".
  - ▶ We will build a solid foundation on the fundamentals of programming, using C++.

# Why C++?

*That was a deliberate policy to have the development of C++ problem-driven rather than imitative. – Bjarne Stroustrup*

- ▶ C++ is very popular and widely used in many application fields, including Finance.
- ▶ Huge demand for C++ skills.
- ▶ Required skill for all, not just programmers.

# Course Structure

- ▶ We will learn C++ using a practical and problem-driven approach.
- ▶ Hands-on and practical course.
- ▶ We learn things step-by-step.

- ▶ We use many tools. Each component is important and connected to one another:
  - ▶ Lectures and notes
  - ▶ Live coding sessions and demos
  - ▶ Discussions: This is a discussion oriented class
  - ▶ Assignments/homework to practice/experiment what we learn.



# Office-hours and TA Sessions and Other Resources

I highly encourage everyone to ask questions.

- ▶ Instructor office-hours:
  - ▶ Fridays before class ( approx. 5 PM CT)
  - ▶ Sundays, 6 PM CT (virtual)
- ▶ TA Sessions (virtual):
  - ▶ Saturdays, 6 PM CT
  - ▶ Weekday - announced later
- ▶ Pl. make use of office-hours, TA sessions as much as possible so we can use the class time efficiently.
- ▶ Other resources:
  - ▶ Email
  - ▶ Canvas discussion forum

# Assignments and Exams

- ▶ 5 Required/graded assignments (each one has equal weight). Graded assignments are to be completed individually.
- ▶ Exams (more details to follow, after week 3):
  - ▶ Mid-term: Feb 3, 6-9 PM CT (Week 5)
  - ▶ Final: March 10, 6-9 PM CT (Week 10)
- ▶ Should use the concepts discussed in class to get full credit.
- ▶ Evaluated based on completeness, correctness, finding the best solution (within what we have discussed), use of correct language constructs, good program design and quality of code.
- ▶ **Academic honesty:** Read: MSFM Academic Honesty Attestation.pdf

# Final Grade

Final grade weights:

1. Graded assignments 30%
2. Mid-term 25% ; Final 45%
3. Bonus points (up to 3 points max) awarded to encourage active student participation and engagement that improve the learning experiences of all students:
  - ▶ Awarded at the instructor's discretion ONLY
  - ▶ Participate class discussions regularly, in a meaningful way.
  - ▶ Provide (meaningful) feedback regularly.
  - ▶ Share relevant interview questions and experiences.

Final Grade cutoffs: per program guidelines

# Reference

- ▶ The Internet
  - ▶ [www.cplusplus.com](http://www.cplusplus.com)
  - ▶ [www.cppreference.com](http://www.cppreference.com)
  - ▶ Microsoft Developer Network [www.msdn.com](http://www.msdn.com)
  - ▶ [www.stackoverflow.com](http://www.stackoverflow.com)
  - ▶ and, more ...

- ▶ Text books NOT required.
- ▶ Following are recommended as sources of further study:
  1. C++:
    - ▶ Bjarne Stroustrup (2014). *Programming Principles and Practice Using C++* (2<sup>nd</sup> Edition). Addison-Wesley.
    - ▶ Bjarne Stroustrup (2013). *The C++ Programming Language* (4<sup>th</sup> Edition). Addison-Wesley.
    - ▶ Stanley Lippman, Josee Lajoie and Barbara Moo (2012). *C++ Primer* (5<sup>th</sup> Edition). Addison-Wesley.
    - ▶ David Vandervoorde and Nicolai Josuttis. *C++ Templates - The complete guide*. Addison-Wesley.
  2. C++ Standard Library:
    - ▶ Nicolai M Josuttis (2012). *The C++ Standard Library* (2<sup>nd</sup> Edition). Addison-Wesley.
  3. Good practices:
    - ▶ Scott Meyers (2005). *Effective C++*. Addison-Wesley.
    - ▶ Herb Sutter (2000). *Exceptional C++*. Addison-Wesley.
  4. Finance:<sup>1</sup>
    - ▶ John Hull. *Options, Futures and Other Derivatives*. Addison-Wesley.

---

<sup>1</sup>This book covers finance theory of applications we use in this course.

## Getting Started

# Programming Environment

- ▶ We use Visual Studio on Windows as our primary programming environment:
  1. Highly user friendly and feature rich programming environment suitable for the beginner and experienced programmer, alike.
  2. Free.
  3. Works on Windows (Most popular O/S among our students).
- ▶ We use CLion (works on Windows/Linux/MacOS) as a secondary environment.
- ▶ Other development environments (e.g. Xcode) are not discussed. If you decide to use any other development environment, you have to learn it on your own.

# Obtaining Software

- ▶ Visual Studio 2022 (Community version):  
<https://visualstudio.microsoft.com/downloads/>
- ▶ CLion: <https://www.jetbrains.com/clion/download/>
- ▶ See *Announcements* for details.
- ▶ Need help? Contact the TAs.



## Getting Started with Visual Studio/CLion

# Getting Started

- ▶ Creating/writing an application using CLion/Visual Studio involve:
  1. Create a project and write code
  2. Organize code (a project/program may use more than one source file)
  3. Build the application
  4. Run the application
  5. Test to make sure it is working correctly
  6. Debug the application
- ▶ We will learn how to do these steps next...

# Getting Started: Hello World Program

- ▶ We will create a very simple project/program to illustrate how to use Visual-Studio/CLion.
- ▶ The program below writes a message, "Hello, World!", to console:

```
#include <iostream>

int main()
{
    std::cout << "Hello, World!" << std::endl;

    return 0;
}
```

# Our First C++ Project

- ▶ First, we create, build and run this program in Visual Studio/CLion.
- ▶ After that, we will explain this program step-by-step and learn C++.
- ▶ Steps for Visual Studio are shown in Appendix A; CLion in Appendix B.

## Exercise

1. Use the steps shown above to create an application/project in Visual Studio/CLion.
2. Code *Hello World* example. **NOTE: C++ is case sensitive**
3. Build the application.
4. Run it.
5. Change the program to print a different message to Console.

# Hello, World Program: Line by Line

```
int main()
```

- ▶ `main()` is a special *function*.
- ▶ It is the program *entry point*.
- ▶ Every application/program should have:
  - ▶ a `main()`.
  - ▶ we can have only one `main()`.
- ▶ We do not pass any arguments to this function.
- ▶ This function returns an integer (`int`) value.
- ▶ Note: We can have two different function signatures for `main` function:
  1. `int main()`
  2. `int main(int argc, char* argv[])`. We will discuss the use of the second function signature later.

- ▶ The line below shows the body of our main function.

```
{  
    std::cout << "Hello, World!" << std::endl;  
  
    return 0;  
}
```

- ▶ A function body starts with { and ends with }.
- ▶ Body has 2 statements; each statement ends with a semicolon (";").
- ▶ First line *writes* a message, "Hello, World!", to console.
- ▶ Here we use some *features* from the C++ Standard Library.
- ▶ After that, the `main()` returns 0 (an integer value)<sup>2</sup>.
- ▶ We use the `return` to return from a function. More on this later.

---

<sup>2</sup>We don't have to return a value in main function. Return 0 (zero) is implied.

## The C++ Standard Library: A First Look



# The C++ Standard Library

- ▶ The C++ Standard Library is a collection of very useful features.
- ▶ Today, we briefly look at the features we use in our first *Hello World* example.
- ▶ The Standard Library is divided into several *sections* (more precisely headers), based on functionality.
- ▶ Each header has a name.
- ▶ We need to *include* the appropriate header in order to have access to its components.
- ▶ The functionality to handle output to console is defined in the `iostream` header.
- ▶ To include it, we type:  

```
#include <iostream>
```
- ▶ We use *angle brackets* to include a header not in our project, e.g. a header from the Standard Library. More on this later.

# The Standard Input and Output

- ▶ Terminology:
  - ▶ Input: a sequence of byte flow coming into the computer memory from an input device (e.g. keyboard, file).
  - ▶ Output: a sequence of byte flow from the memory to an output device (e.g. console, file).
  - ▶ Stream: a sequence of characters (bytes) read from an input device or written to an output device.
- ▶ The `iostream` section of defines objects to read and write a *stream* to *Standard I/O* channels:
  - ▶ `cin`
  - ▶ `cout`
  - ▶ and more..

# Standard Output

`cout` (pronounced as *see-out*):

- ▶ Known as the **standard output**.
- ▶ Attached to the standard output device, which usually is the console
- ▶ Uses the stream insertion operator, `<<`:  
`cout << "Hello, World";`

# Standard Input

`cin` (we pronounce it as *see-in*):

- ▶ Known as the **standard input**
- ▶ Attached to the standard input device, which usually is the keyboard
- ▶ Uses the stream extraction operator ( `>>` ), to read a stream from the standard input:

```
int x;  
cin >> x;
```

# Namespace

- ▶ Namespaces allow us to group functions and classes using a name.
- ▶ Names (functions, classes) in the C++ Standard Library are defined in the `std` namespace.
- ▶ We have to use the fully qualified name to use them.
- ▶ Example: We saw the use of `std::cout`.
  - ▶ `std` is the namespace.
  - ▶ `::` is the scope resolution operator in C++.
  - ▶ `cout` is the object in this case.

# Using Namespaces

- ▶ Having to write the namespace every time we use a function is a bit cumbersome.
- ▶ There are different ways to overcome this problem.
- ▶ Technique 1: qualify individually, e.g. `cout` to mean `std::cout`

```
#include <iostream>
using std::cout;
using std::endl;

int main()
{
    cout << "Hello, World!" << endl;
}
```

- ▶ Technique 2: allows us to access all names from `std` namespace

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

int main()
{
    cout << "Hello, World!" << endl;
}
```

- ▶ If we're not careful, the second technique can lead to name clashes in large programs (where the same name is used in more than one namespace, example later).
- ▶ I encourage everyone to use the first technique.
- ▶ Note:
  - ▶ I may not show every namespace in subsequent slides after they are introduced.
  - ▶ E.g. From now on, I will just write `cout` (without `using std::cout`).

## Data Types



# Data Types

- ▶ We write programs to handle/manipulate data.
- ▶ We use variables to store data.
- ▶ A program need to store different *kinds/types* of data.
- ▶ In C++:
  - ▶ Every variable has to have a type.
  - ▶ The type of the variable has to be declared before it is used.
  - ▶ Once a type is assigned to a variable, it cannot be changed.
- ▶ Data types in C++ belong to two main categories:
  1. Fundamental data types
  2. User defined data types
- ▶ Fundamental data types are defined in the C++ language.
- ▶ C++ also allows the user to define types (classes).

## Fundamental Data Types

# Numerical Types

- ▶ Numerical values can be whole or real numbers: e.g. 1 or 1.2
- ▶ For whole numbers we have several types:
  - ▶ short: uses 2 bytes
  - ▶ int: uses 4 bytes
  - ▶ long: at least 4 bytes
- ▶ Signed and unsigned:
  - ▶ unsigned types can store positive numbers only
  - ▶ signed types can store both positive and negative numbers

Type	Size (Bytes)	Value Range
unsigned short	2	0 to 65,535
short	2	-32,768 to 32,767
unsigned int	4	0 to 4,294,967,295
int	4	-2,147,483,648 to 2,147,483,647

- ▶ For real numbers we have two types (in C++):
  - ▶ float : uses 4 bytes
  - ▶ double : uses 8 bytes
- ▶ As a result, max and min values each can store are different.

Type	Size (Bytes)	Value Range
float	4	$\pm 3.4 E^{\pm 38}$
double	8	$\pm 1.7 E^{\pm 308}$

- ▶ We can use them to store both negative and positive real numbers.
- ▶ Remember: only a very few real values can be stored exactly.

## More Types

- ▶ Type to hold a true/false (flag) value
  - ▶ `bool`
- ▶ Type to hold a single character
  - ▶ `char`
- ▶ Reference: <https://docs.microsoft.com/en-us/cpp/cpp/data-type-ranges?view=msvc-170>
- ▶ Type to store a string of characters
  - ▶ `std::string`
  - ▶ `std::string` is NOT a fundamental data type
  - ▶ defined in the C++ Standard Library
  - ▶ defined in the string header (`<string>`)

## C++ Operators

# Operators for Fundamental Data Types

Listed below are some operators defined for fundamental types:

- ▶ Assignment operator (=) :

```
a = 5;      //a is an int
b = 1.5f;   //b is a float
c = 1.4;    //c is a double
d = 'a';    //d is a char
e = true;   //e is a bool
```

- ▶ Arithmetic operators (+, -, \*, /) :

```
a = a + 1;
b = b - 1.2f;
c = c * 2.1 ;
c = c / 2.2;
```

- ▶ Comparison operators (<, >, <=, >=):

```
a > 3;
a < 3;
a >= 3;
a <= 3;
```

Comparison operators return a bool (true/false) value.

- ▶ Equality (==):

`a == 3;`

- ▶ Non Equality (!=) :

`a != 3;`

- ▶ Logical AND (&&) : Returns true if both operands are true; false otherwise<sup>3</sup>

`a == 3 && d == 'a';`

- ▶ Logical OR (||) : Return true if either or both operands is true; false otherwise

`a == 3 || d == 'a';`

- ▶ Modulo % (to find the remainder from integer division):

`int x = 7 % 3;`

---

<sup>3</sup>[https:](https://en.wikipedia.org/wiki/Truth_table#Logical_conjunction_(AND))



## Operators: Short-cuts

We have some short-cuts; e.g. For an `int` value `i`:

- ▶ Increment:

```
i++;      // same as, i = i+1;  
i += 17;  // same as, i = i+17;
```

- ▶ Decrement:

```
i--;      // same as, i = i-1;  
i -= 21;  // same as, i = i-21;
```

- ▶ Multiplication:

```
i *= 13;  // same as, i = i*13;
```

- ▶ Division:

```
i /= 42;  // same as, i = i/42;
```

# Prefix and Postfix

C++ supports prefix and postfix increment and decrement operators.

- ▶ Postfix:

- ▶ increment x and return old x:

```
int x = 3;  
int y = x++;
```

- ▶ now, y == 3 and x == 4

- ▶ Prefix:

- ▶ increment x and return new value

```
int x = 3;  
int y = ++x;
```

- ▶ now, y == 4 and x == 4

# Functions

# C++ Functions

- ▶ We saw the `main()` function – every application has to have a `main()`.
- ▶ We do NOT write the entire program in the main function.
- ▶ It is easier to solve a problem by breaking it up into smaller tasks/functions:
  - ▶ We write a function to do a certain task.
  - ▶ Combine functions to solve the problem.
- ▶ Using functions has several advantages.

- ▶ First, let's learn how to write functions in C++.
- ▶ Example: we can write a function to display a greeting message on the screen:

```
void DisplayGreeting()  
{  
    cout << "Hello World!" << endl;  
}
```

- ▶ A C++ function:
  - ▶ has a name
  - ▶ takes zero or more arguments
  - ▶ does something in the function body
  - ▶ returns some value of a certain type
  - ▶ if a function does not return anything, its return type is `void`

## Functions: Example 1

- ▶ Our first function writes the greeting, "Hello World!" to the screen:

```
void DisplayGreeting()
{
    cout << "Hello World!" << endl;
}
```

- ▶ We can call this function from the `main()` or any other function:

```
int main()
{
    DisplayGreeting();
}
```

## Functions: Example 2

- How would you write a function to add two integers?

```
int add(int a, int b)
{
    return a+b;
}
```

```
int main()
{
    int result = add(2, 3);

    cout << " Result :" << result << endl;
}
```

# Functions: Advantages

1. Better/simpler solution: divide the problem into smaller tasks.
2. Better code structure: improve readability/clarity.
3. Reuse: once a function is written, we can use it any number of times.
4. Maintainability: we write a function once, if we want to make a change, fix a bug, we can do it just in one place.
5. Type safety (avoiding type related errors):
  - ▶ Types of arguments must be compatible with the types declared in the function.
  - ▶ Program will not compile if there's a mismatch.



# Parameter Overloading

- ▶ Overloading allows 2 or more functions to have the same name:
  - ▶ they must have different input argument **types**
  - ▶ they cannot differ only by the return type
- ▶ Allows us to write more than one method to do some task in a different way, using different inputs.
- ▶ For example, we can write another add() function as:

```
double add(double a, double b)
{
    return a+b;
}
```

# Implicit Type Conversions

- ▶ Suppose we have the following function (i.e. it is the only add function we have):

```
double add(double a, double b)
{
    return a+b;
}
```

- ▶ What happens if we use it to add two integer values:

```
int main()
{
    int x = 2;
    int y = 3;
    cout << add(x, y) << endl;
}
```

- ▶ Question: does this program compile (build)?

- ▶ Yes, this program compiles:
  - ▶ Compiler converts the int to double (implicitly) in this case.
  - ▶ Size of int: 4 bytes; double: 8 bytes.
  - ▶ We can safely store any int value in a double.
- ▶ Conversion from a smaller (4 byte) to a larger (8 byte) type is called a *widening, or, promotion*
- ▶ Promotions are safe implicit conversions.

- ▶ Now, suppose we have the following function (i.e. it is the only add function we have):

```
int add(int a, int b)
{
    return a+b;
}
```

- ▶ What happens if we use it to add two double values:

```
int main()
{
    double x = 2.2;
    double y = 3.3;
    cout << add(x, y) << endl;
}
```

- ▶ Will this program compile (build)?

- ▶ Yes, this program also compiles:
  - ▶ Compiler converts the double to int values implicitly in this case
  - ▶ Size of double: 8 bytes; int: 4 bytes
  - ▶ We cannot safely convert every double value to an int
- ▶ Conversion from a larger type to a smaller type is known as *narrowing*.
- ▶ Narrowing can be dangerous, and results in a build *Warning*.
- ▶ Why is it not an error?

- ▶ Why it is not an error:
  - ▶ C++ expects users to know what they are doing.
  - ▶ You can easily shoot yourself in the foot if we're not careful.
- ▶ Lesson: Pay Attention to Build Warnings
- ▶ We can use project properties to treat warnings as errors (more on this later).
  - ▶ Visual Studio: Properties ⇒ C/C++ ⇒ General ⇒ Treat Warnings as Errors ⇒ Yes
  - ▶ CLion:
    - ▶ CLion compiler-flags: <https://www.jetbrains.com/help/clion/cmake-profile.html#compiler-flags>
    - ▶ gcc options: <https://gcc.gnu.org/onlinedocs/gcc/Option-Summary.html>
    - ▶ E.g. for gcc compiler, add the following line in your CMakeLists.txt script:  
`set(CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} -Werror -Wconversion")`

# Casting

- ▶ Explicit type conversions are known as *casting*.
- ▶ C++ supports several types of casts (known as *named casts*).
- ▶ We'll look at `static_cast` now:
- ▶ What's the result from integer division below?

```
int a = 3, b = 2;  
cout << a / b << endl;
```

- ▶ We must explicitly convert at least one operand to a real value to get the correct answer.
- ▶ `static_cast`:
  - ▶ Used to force conversions, e.g. `int`->`double`  

```
int a = 3; int b = 2;  
cout << static_cast<double>(a) / b << endl;
```
  - ▶ `static_cast` returns a double value in this example.



# The const keyword

- ▶ The `const` keyword is used to define a constant value (i.e. value does not change):
  - ▶ clearly communicates the intent to the reader
  - ▶ prohibits unintentional changes
- ▶ If you try to change the value of a `const` member, the compiler catches it.  
`const int pi = 3.14;`  
`pi = 4.5;` <= This is an error, compiler will catch this error
- ▶ Following notation is also equally correct:  
`int const pi = 3.14;`
- ▶ This is another example of the benefits of type safety in C++.
- ▶ We will discuss more use cases of the `const` keyword later in the course.

# Type Alias

- ▶ We use type aliasing to define more meaningful names for existing types (i.e. type aliasing) to make code:
  1. clear and readable
  2. less error prone
- ▶ Consider the function declaration below which calculates the Black Scholes price of an Option.

```
double CallPrice(double s,  
                 double k,  
                 double r,  
                 double v,  
                 double t);
```

- ▶ Error prone: it is not clear what the arguments are
- ▶ Instead, we can use a type alias:
  1. typedef
  2. using
- ▶ typedef:

```
typedef double StockPrice;  
typedef double Strike;  
typedef double Expiration;  
typedef double Rate;  
typedef double Volatility;
```

► using:

```
using StockPrice = double;  
using Strike = double;  
using Expiration = double;  
using Rate = double;  
using Volatility = double;
```

► Now we can write:

```
double CallPrice(StockPrice s,  
                 Strike k,  
                 Expiration t,  
                 Rate r,  
                 Volatility v);
```

► This is an example of writing "clear and readable" code.

# Comments

- ▶ We can write comments using C style or C++ style.

- ▶ C Style:

```
/* This is a C-style comment.  
   Useful when we comment ...  
   more than one line.  
*/
```

- ▶ C++ Style:

```
// This is a C++ style comment  
// Each line has to be commented ..  
// when we use this style.
```

- ▶ Use good comments to improve readability and clarity.

## .cpp Source and \*.h Header Files

- ▶ In practice we never write all code in one file:
  - ▶ Difficult to read a very large file (with thousands of lines of code)
  - ▶ Doesn't promote reuse
  - ▶ Build process takes longer
- ▶ Solution is to use separate files:
  - ▶ .cpp source files
  - ▶ .h header files

- ▶ Let's use the add functions as an example:
  - ▶ We will implement the `add()` functions in a separate file, let's name it `Add.cpp`.
  - ▶ When we move the add functions to a new file our program won't build.
  - ▶ The `main()` function doesn't know about the `add()` functions.
  - ▶ We need to declare a function before we can use it in another source file.
  - ▶ We could add the following two declarations before we use them in `main`:

```
int add(int, int);  
double add(double, double);
```
  - ▶ It works, but this solution is not ideal: it requires us to have these declarations in every file we use them.

- ▶ Proper C++ way:
  - ▶ Write the function declarations in a header file (e.g. Add.h):  
`int add(int, int);`  
`double add(double, double);`
  - ▶ *Include* it in source files, using `#include` preprocessor directive:  
  
`#include "Add.h"`
  - ▶ We use `" "` to include headers in the same project.

# C++ Build Process

C++ build process uses several steps:

## 1. Preprocessing:

- ▶ Uses the preprocessor
- ▶ The preprocessor handles the preprocessor directives (e.g. `#include`)
- ▶ The output of this step is a C++ file without preprocessor directives

## 2. Compiling:

- ▶ Uses the compiler.
- ▶ Checks syntax.
- ▶ Checks type safety.
- ▶ Does other things (e.g. implicit type conversions). More on this later.
- ▶ Can do computations (we won't discuss this topic in this course, but we will talk about it in my HPC course).
- ▶ Each `.cpp` file is compiled to create an object file `(.obj)`.

## 3. Linking:

- ▶ Uses the linker
- ▶ Creates the executable `(.exe)` file using the object files



- ▶ We can have build errors in each stage.
- ▶ Having an idea about each stage will help us fix build problems (more on this later).

# Lecture 1: Key Points

- ▶ Creating a project, using .h and .cpp files to structure code
- ▶ Fundamental types and operations.
- ▶ Functions.
- ▶ Writing clear/readable code.
- ▶ Build process - first look.
- ▶ Understand (some) things compiler can do.
- ▶ Should pay attention to warnings.
- ▶ How to submit an assignment.

# Assignment 0 (Submission Optional)

## Goals:

1. To make sure everyone has a C++ development environment.
2. Learn the proper structure of a C++ program.
3. Use fundamental types, operators and functions.
4. To make sure everyone knows how to submit an assignment.

## Problem:

1. Write a function to add two integers.
2. Read any two integer values from the keyboard.
3. Add them using the function you wrote.
4. Write the result to console.

## Submission steps:

- ▶ Clean the solution using *Clean Solution* under *Build* tab (in Visual Studio/CLion).
- ▶ Compress (e.g. zip on Windows) the folder containing the project.
- ▶ Use *Assignments* -> *Assignment 0* link in canvas to upload the compressed/zipped file.

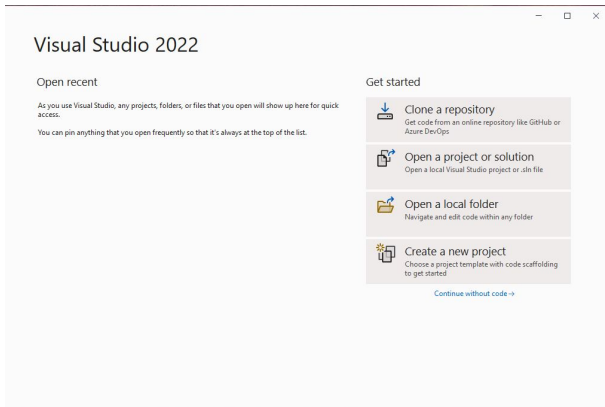
- ▶ Pl. use following guidance on how to name your solutions:
  - ▶ Visual Studio users: Name the solution using the following format. "Lastname\_Firstname\_Assignment#".
  - ▶ CLion Users: Name the project using the following format. "Lastname\_Firstname\_Assignment#" .
- ▶ This is not a requirement, but it helps TAs sort all of the source code that gets submitted.

- ▶ Individual Assignment.
- ▶ Due: January 13 by 6 PM (Central Time).
- ▶ Points: 0 (Zero)

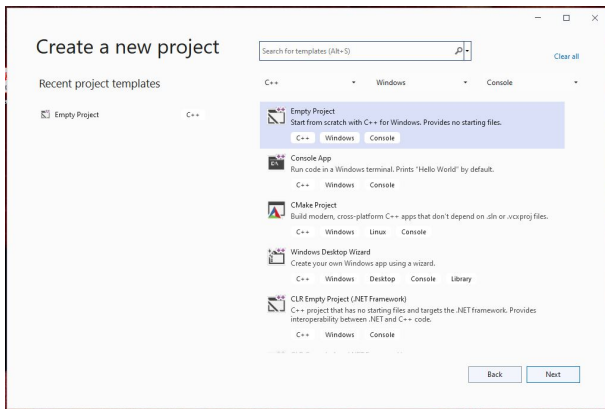
## Appendix A: Getting Started with Visual Studio

# Steps

- ▶ Step-by-step instructions to create an application are shown below.
- ▶ Start Visual Studio. Select *Create a new project*.



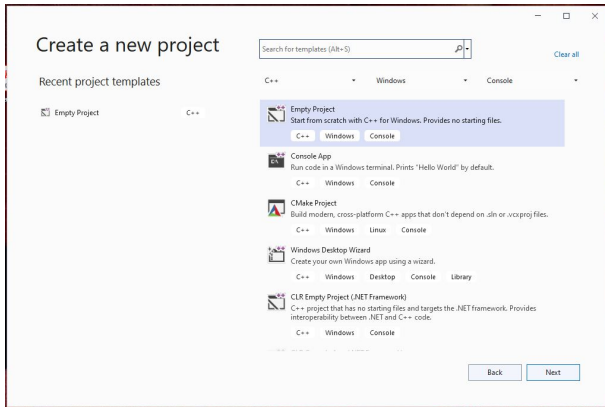
- Select "C++" -> first and then "Empty Project"



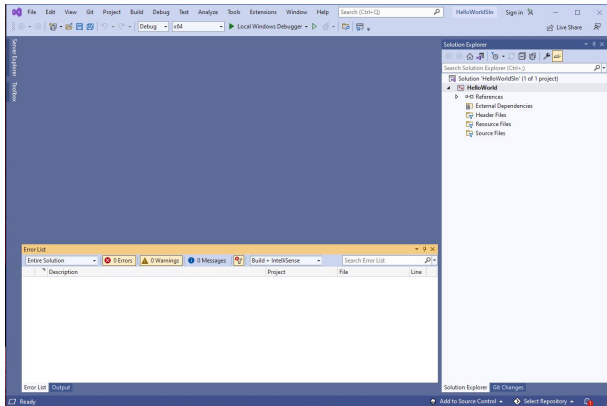


Enter:

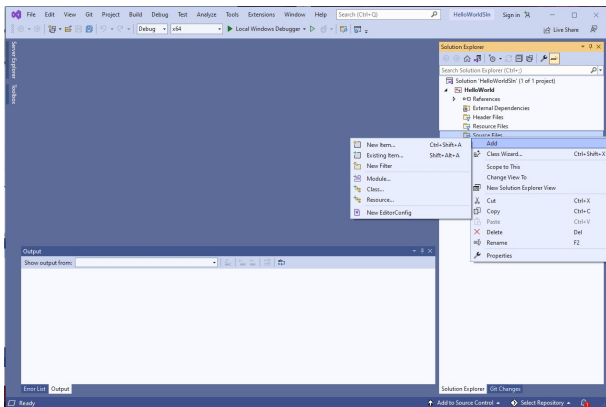
1. (project/application) name
2. directory location for the project
3. solution (place holder for one or more projects) name for the project.



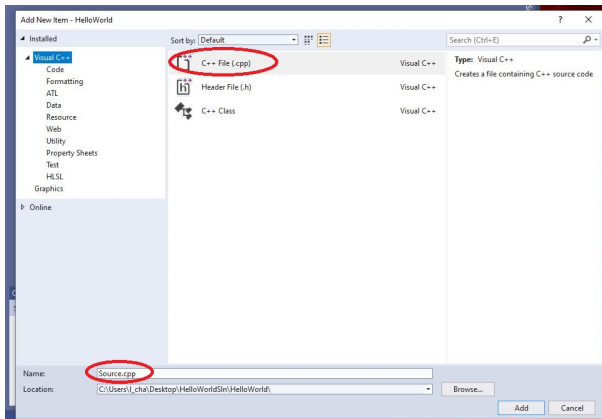
- We have a new Solution with 1 project.



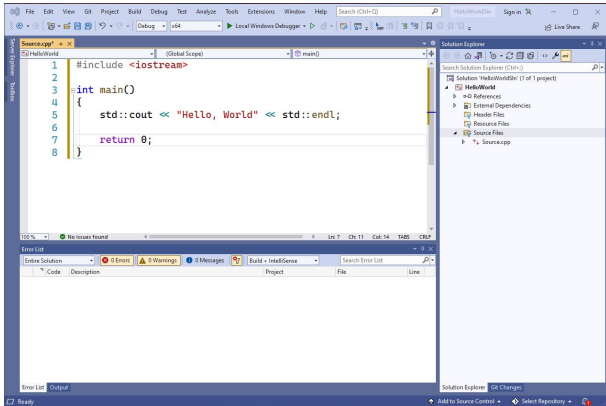
- ▶ Let's add a new source file to the project so we can write code.
- ▶ Right click on the "Source Files" folder or the Project name on the Solutions Explorer.
- ▶ Select, Add -> New Item



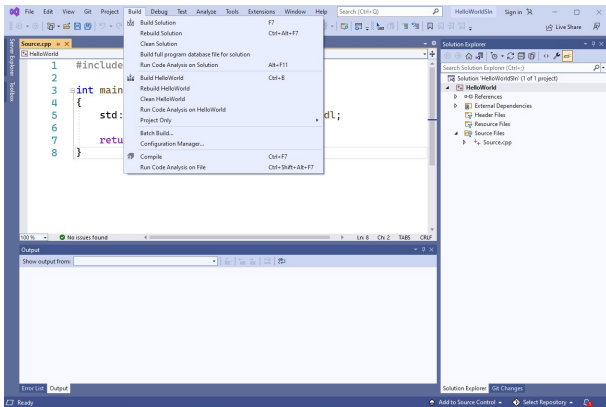
- ▶ Select **C++ File**
- ▶ In large projects we name the source files appropriately (more on this later). For this tutorial the default of Source.cpp is just fine. Click *Add* button.
- ▶ Now you have a empty cpp file.



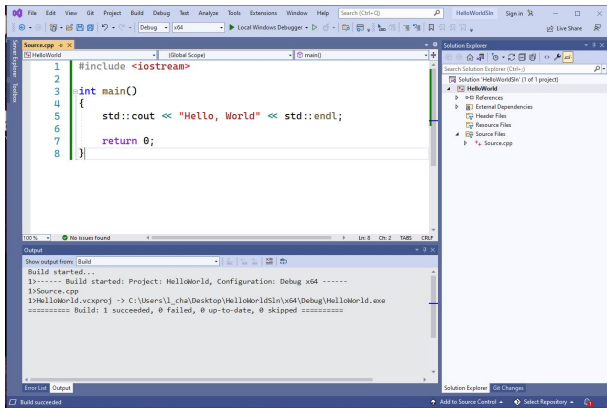
- ▶ You may type code, or copy and paste code below:
- ▶ If you type, note that C++ is a case sensitive language.



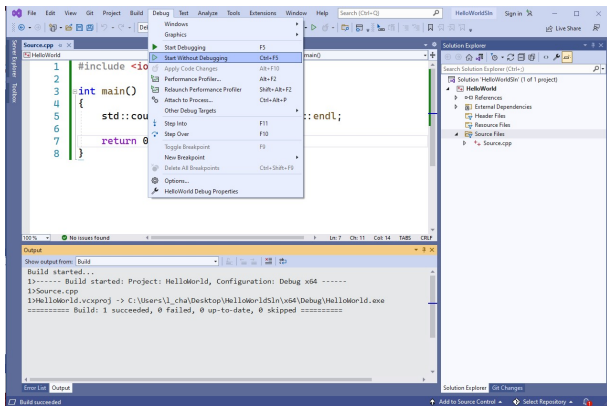
- Next step is to build the project. Select, *Build* → *Build Solution*.



- Make sure the program builds successfully.

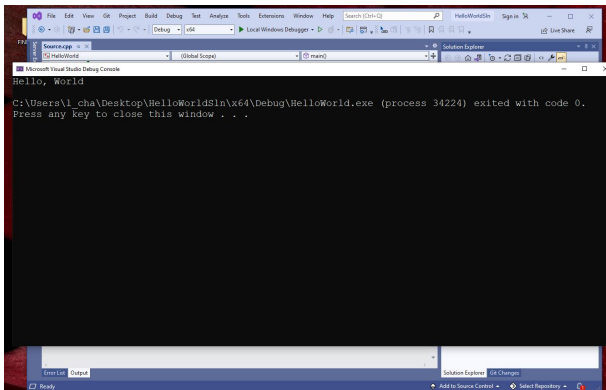


- To run, select *Debug* -> *Start Without Debugging*



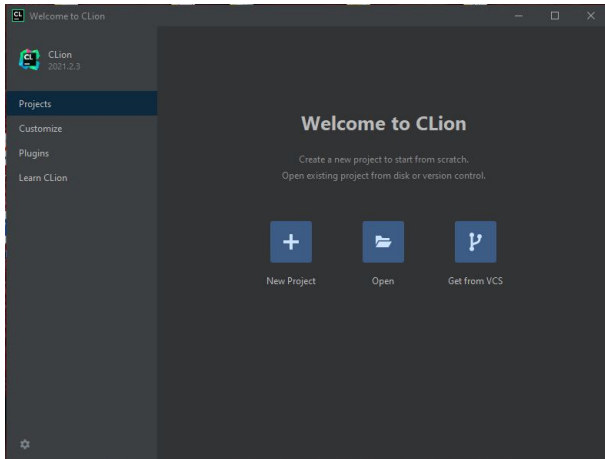


- ▶ You'll see the greeting message on Console.

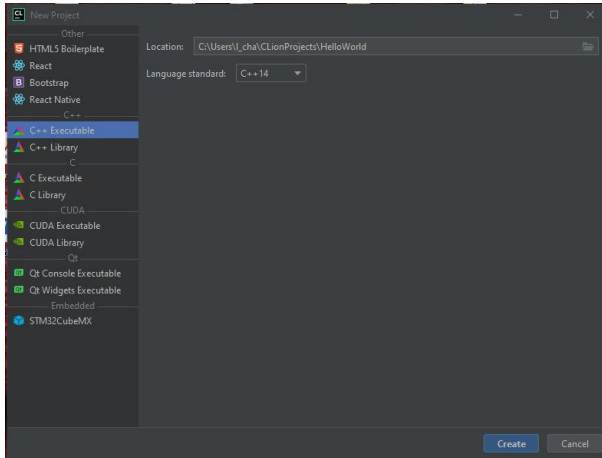


## Appendix B: Getting Started with CLion

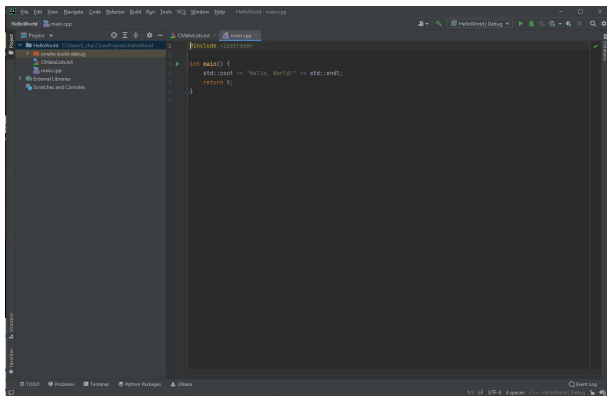
► Open CLion IDE.



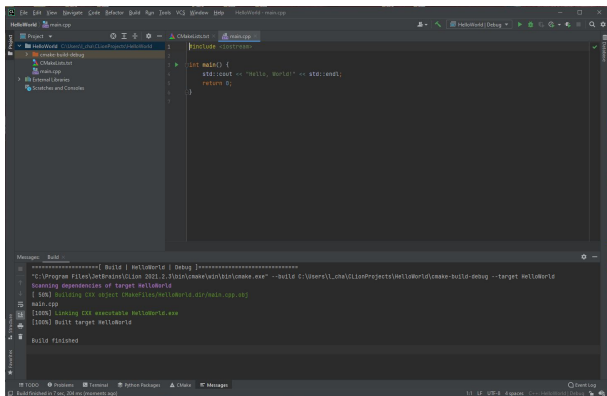
- ▶ Select “C++ Executable” project type.
- ▶ Select a location for the project.



- ▶ CLion by default includes code for Hello-World example.



- Build the project. Make sure it builds.



The screenshot shows a CMake IDE window titled 'HelloWorld - main.cpp'. The interface includes a menu bar (File, Edit, View, Debugger, Build, Run, Tools, Window, Help), a toolbar, and a sidebar with a 'Project' view. The 'Project' view shows a tree structure with 'HelloWorld' as the root, containing 'CMakeLists.txt', 'main.cpp', 'HelloWorld', and 'Scopes and Consoles'. The 'main.cpp' file is open in the editor, showing the following code:

```
#include <iostream>

int main() {
    std::cout << "Hello, World!" << std::endl;
    return 0;
}
```

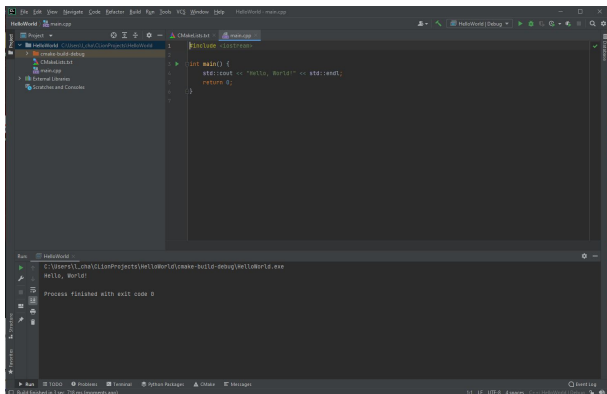
Below the editor is a 'Messages' panel with a 'Build' tab. It displays the output of the build process:

```
===== Build | HelloWorld | Debug |=====
"C:\Program Files\JetBrains\CLion 2021.2.3\bin\cmake\win\bin\cmake.exe" --build C:\Users\l_t_cha\CLionProjects\HelloWorld\cmake-build-debug --target HelloWorld
Scanning dependencies of target HelloWorld
[ 50%] Building CXX object CMakeFiles/HelloWorld.dir/main.cpp.obj
main.cpp
[100%] Linking CXX executable HelloWorld.exe
[100%] Built target HelloWorld

Build finished
```

The status bar at the bottom indicates 'Build finished in 7 sec, 204 ms (Documents app)' and shows a 'Debug' button.

- ▶ And, run it.
- ▶ You'll see the greeting message on Console.



```
1 #include <iostream>
2
3 int main() {
4     std::cout << "Hello, World!" << std::endl;
5     return 0;
6 }
```

Run HelloWorld

```
C:\Users\l_csa\CLionProjects\HelloWorld\cmake-build-debug\HelloWorld.exe
Hello, World!

Process finished with exit code 0
```

- ▶ Steps above illustrate steps to get started.
- ▶ I will illustrate more advanced examples later but the amount of time we spend on CLion is limited.
- ▶ For more info, use: <https://www.jetbrains.com/help/clion/clion-quick-start-guide.html>



## Appendix C: Binary Representation

# Storing Values in Computer Memory

- ▶ We use bits store data; a bit stores a binary (0/1) value.
- ▶ A computer memory is organized as units of bytes.
- ▶ One byte is 8 bits.
- ▶ Suppose we have 3 bits and we want to store an integer value.  
We can store: 000, 001, 010, 011, 100, 101, 110, 111.
- ▶ First, let's look at a familiar example using decimal representation:  $147 = 1 * 10^2 + 4 * 10^1 + 7 * 10^0$
- ▶ Similarly, we can use the binary representation:

Binary	Decimal Value
000	$0 * 2^2 + 0 * 2^1 + 0 * 2^0 = 0$
001	$0 * 2^2 + 0 * 2^1 + 1 * 2^0 = 1$
010	$0 * 2^2 + 1 * 2^1 + 0 * 2^0 = 2$
011	$0 * 2^2 + 1 * 2^1 + 1 * 2^0 = 3$
100	$1 * 2^2 + 0 * 2^1 + 0 * 2^0 = 4$
101	$1 * 2^2 + 0 * 2^1 + 1 * 2^0 = 5$
110	$1 * 2^2 + 1 * 2^1 + 0 * 2^0 = 6$
111	$1 * 2^2 + 1 * 2^1 + 1 * 2^0 = 7$

- ▶ How do we store a negative value?
- ▶ We use the leftmost bit to store the sign.
- ▶ This reduces the maximum value we can store.
- ▶ Usually signed values are stored using what's known as *twos-complement* method<sup>4</sup>.

---

<sup>4</sup>[http://en.wikipedia.org/wiki/Two's\\_complement](http://en.wikipedia.org/wiki/Two's_complement)

- ▶ How do we store fractions?
- ▶ Again, let's look at a familiar example:  
 $0.147 = 1 * 10^{-1} + 4 * 10^{-2} + 7 * 10^{-3}$
- ▶ Similarly, we can use the binary representation: Let's see what happens if we were to use 2 bits to store the fractional part

Binary	Decimal Value
00	$0 * 2^{-1} + 0 * 2^{-2} = 0$
01	$0 * 2^{-1} + 1 * 2^{-2} = 0.25$
10	$1 * 2^{-1} + 0 * 2^{-2} = 0.50$
11	$1 * 2^{-1} + 1 * 2^{-2} = 0.75$

- Suppose we have 3 bits:

Binary	Decimal Value
000	0
001	0.125
010	0.25
011	0.375
100	0.5
...	...

- Homework: Complete the table above.

- ▶ Observations:
  1. Very few values can be exactly represented as a real value
  2. How close can we get depends on the number of bits
- ▶ Note: Real values are stored using an exponent and a mantissa:
  - ▶  $\text{value} = \text{mantissa} * 2^{\text{exponent}}$
  - ▶ how many bits are reserved for the mantissa part and exponent part are defined (by the IEEE standard).