

kintsugi-stack-dsa-cpp: CPP

"C makes it easy to shoot yourself in the foot; C++ makes it harder, but when you do it blows your whole leg off."





- Author: [Kintsugi-Programmer](#)

Disclaimer: The content presented here is a curated blend of my personal learning journey, experiences, open-source documentation, and invaluable knowledge gained from diverse sources. I do not claim sole ownership over all the material; this is a community-driven effort to learn, share, and grow together.

Table of Contents

- [kintsugi-stack-dsa-cpp: CPP](#)
 - [Table of Contents](#)
- [Chapter 1: C++ Introduction](#)
 - [1. Series Announcement and Introduction](#)
 - [2. Instructor and Channel Background](#)
 - [3. Course Logistics and Engagement](#)
 - [4. Why C++? Language Features and Use Cases](#)
 - [4.1. Reason for Popularity: Academics](#)
 - [4.2. Platform Independence](#)
 - [4.3. Efficiency and Large Scale Applications](#)
 - [4.4. Object Oriented Programming \(OOP\)](#)
 - [4.5. Statically Typed](#)
 - [4.6. Speed and System Proximity](#)
 - [4.7. Abstraction Layer and Use in Other Languages](#)
 - [4.8. Inner System Knowledge](#)
 - [4.9. Pointers and Manual Memory Management](#)
 - [5. The Journey of C++: Bjarne Stroustrup](#)
 - [5.1. The Creator](#)
 - [5.2. Birth of the Language](#)
 - [5.3. C++ Major Versions](#)
 - [6. C++ Environment Setup](#)
 - [6.1. Compiled Language Concept](#)
 - [6.2. Official Documentation](#)
 - [6.3. Compilers and Tools](#)
 - [6.4. Setting up VS Code \(Step-by-Step\)](#)
 - [6.5. Alternative Code Environments](#)
 - [7. Next Steps](#)
 - [Code](#)
- [Chapter 2: Anatomy of the Hello World C++ Program](#)
 - [1. Introduction and Context](#)
 - [1.1 Challenges in Programming](#)
 - [1.2 Video Goals and Best Practices](#)

- 2. Program Execution Flow
 - 2.1 OS Interaction and Executables
 - 2.2 The Starting Point: The `main` Method
- 3. Anatomy of the C++ Program Lines
 - 3.1 Preprocessor Directives: `#include <iostream>`
 - 3.2 Whitespace and Compiler Smartness
 - 3.3 Namespaces: `using namespace std;`
 - Example of a Custom Namespace
 - Real-World Examples of Namespaces
 - 3.4 Methods for Using Namespaces
 - Method 1: Using the Scope Resolution Operator (`::`)
 - Method 2: Importing Specific Elements with `using`
 - 3.5 The `main` Function and Return Types
 - Functions and Functionality
 - Rules for Function Definition
 - Return Values and Exit Codes
 - 3.6 Output Operators and Semicolons
 - Semicolon Rule
- 4. Summary of Key Concepts
- Code
- Chapter 3: Variables and Constants
 - I. Introduction and Series Context
 - Video Goals and Pacing
 - II. Setting Up the Code Structure
 - File and Folder Naming
 - Why `main.cpp`?
 - III. Basic C++ Program Structure
 - Code Block: Basic C++ Program
 - IV. Understanding and Using Comments
 - Purpose of Comments
 - Compiler Interaction with Comments
 - Types of Comments and Shortcuts
 - Professional Commenting Best Practices
 - White Spaces
 - VS Code Line Manipulation
 - V. The Fundamental Role of Data
 - Data Storage
 - Data Processing
 - VI. Variable and Constant Declaration
 - Variables
 - 1. Declaration (The Syntax)
 - 2. Initialization (Assigning Value)
 - 3. Modification
 - Identifiers (Variable Naming Rules)
 - Keywords
 - Allowed Naming Conventions

- What is NOT Allowed
- Constants
 - 1. Declaration using `const`
 - 2. Attempting to Modify a Constant
- Left-Hand Side value  & Right-Hand Side value 
 - L-value (The Mailbox) 
 - R-value (The Letter) 
 - The Golden Rule: Assignment
 - Why This Mattered in Your Last Example
- VII. Next Steps in Programming
- Code

Chapter 1: C++ Introduction

1. Series Announcement and Introduction

- Resource : [Chai aur C++](#)

In this series, **we will write a lot of C++ code**, understand it, and go in-depth. C++ is a very fun and interesting language.

By the time the series concludes, you will be experts in C++. You will enjoy writing code. Crucially, you will understand the **code flow, architecture, and how to convert thoughts into code**.

2. Instructor and Channel Background

The instructor's name is Hitesh. He has been coding and teaching for the last 12–15 years. He has taught and explained code to millions (lakhs) of students. He has worked at several companies (though he will not display FAANG logos, he has worked extensively). Students are in good hands for learning C++.

3. Course Logistics and Engagement

The videos will be long. They will include:

- Stories.
- Projects.
- Assignments and questions.
- A lot of content will be included, as always.

The first video covers the initial story, installation steps, and other miscellaneous topics. Today, the instructional team is sitting with **cold tea** (*thandi chai*).

4. Why C++? Language Features and Use Cases

C++ is already a very popular language.

- Reason for Popularity:
 - Academics
 - Platform Independence

- Efficiency and Large Scale Applications
- Object Oriented Programming (OOP)
- Statically Typed
- Speed and System Proximity
- Abstraction Layer and Use in Other Languages
- Inner System Knowledge
- Pointers and Manual Memory Management

4.1. Reason for Popularity: Academics

The number one reason for C++'s popularity is **academics**. When academic curricula were designed many years ago, the structure followed the historical timeline of languages. They started with C, then C++, and then languages like Java (an object-based language popular at the time) were added. PHP is also included in some curricula. Many college students and even school students study C++.

4.2. Platform Independence

C++ is **purely platform independent**.

- The code can execute anywhere.
- Libraries and binaries are required for compilation.
- Once the code is compiled, the executable code (binaries) can be run anywhere.
- Most software seen on Windows, such as **.exe** files, are easily built using C++.

4.3. Efficiency and Large Scale Applications

- C++ is used for building large-scale applications.
- Building software in C++ is **memory efficient** than other Langs, js etc.
- It allows for efficient memory management.
- It is suitable for **high-end applications**.

4.4. Object Oriented Programming (OOP)

C++ was specifically built as an Object Oriented language.

- The object-oriented path originated from a **PhD thesis**.
- C++ is one of the first languages that properly defines object-oriented concepts.
- It provides all the base structures **openly**.
- It does not hide or abstract away many concepts, leading to strong foundational knowledge in OOP.

4.5. Statically Typed

C++ is a **statically typed** language.

- In statically typed languages, the data type of a variable is specified beforehand.
- This is analogous to filling out a form, where you know specifically where to write numbers or words.
- Knowing the data type (number, string, letter) beforehand simplifies the work.
- This leads to **fewer mistakes** and increases predictability.

4.6. Speed and System Proximity

C++ is one of the **fastest languages** because it operates very close to the system.

- *Note:* If speed is the only reason for learning C++, Assembly language is faster.
- C++ remains closer to the system but offers enough abstraction, making it comparatively easier to write code than Assembly.
- Modern APIs and libraries are often built using C++.

Many Games , Softwares(Like Dropbox, Adobe Ps,etc.) and Even Servers Are built in C++ and are Very Efficient !!!

C++ is completly capable to interact with MordernDBs like MongoDB, Postgres, etc., Almost Most Have Drivers in C++.

4.7. Abstraction Layer and Use in Other Languages

C++ was used in genesis of Many Many stuff. C++ is often abstracted and used to design APIs for other, higher-level languages.

Abstracted Language	C++ Role	Details
Python	Underlying Engine	Major Machine Learning (ML) libraries like NumPy and Pandas were originally designed in C++. Programmers preferred an easier language (Python), so an exposure layer was created to allow them to use these libraries via Python.
JavaScript (JS)	V8 Engine	The original JavaScript V8 engine is built in C++. JS is used because it is easier.
Mobile Applications (React Native)	Core Functionality	A large chunk of React Native mobile application development is built entirely in C++.

4.8. Inner System Knowledge

Learning C++ provides **inner system knowledge**.

- It helps you understand what happens to your data when it enters the memory.
- This system understanding is the primary reason for learning C++, not just speed.
- This knowledge helps in extraordinary cases, such as performing **inner optimization** or tweaking at a large-scale company.
- C++ is the language where hardware, graphics, and device drivers are available and written, allowing users to see how they are written and installed.

During Big Projects and Systems, Having Good Background of C++ helps in tweaking, which mayn't be possibly done by other High-level frameworks !!!

4.9. Pointers and Manual Memory Management

C++ is one of the languages that allows **direct, manual memory management**.

- Other languages typically use automated Garbage Collection (where unused variables are removed automatically).
- C++ gives the capability to **manually** manage memory.
- It was the first language to introduce the concept of **Pointers**.
- Pointers provide a direct memory reference.
- This allows the developer to manipulate or read data directly in memory, bypassing the variable pathway.
- This reduces an abstraction layer, which improves system understanding and allows for better command over abstracted languages.

5. The Journey of C++: Bjarne Stroustrup

5.1. The Creator

We must acknowledge **Bjarne Stroustrup**, the creator of C++.

- He has created a great language and done extensive work.
- He has videos on YouTube and attends conferences (e.g., in Europe).
- He still provides services and his homepage is available.
- He publishes content like "Tour of C++" and is active in development.

5.2. Birth of the Language

Stroustrup was heavily influenced by the Object Oriented Programming paradigm and wanted to bring its principles to commercial software development.

1. **Initial Attempt (Simula):** During his PhD, Stroustrup used the language **Simula**. His major thesis portion focused on attaching OOP principles to Simula.
2. **Failure:** This thesis failed. When OOP principles were attached to Simula, the speed dropped dramatically, making the language almost unusable.
3. **New Idea (C Integration):** Stroustrup decided to integrate OOP principles with the existing C language(like Right now TypeScript is integrating with JavaScript). He took the code base and detached some parts of Simula.
4. **Early Language and Compiler:** Object Oriented principles were integrated with C.
 - A new compiler named **Cfront** (pronounced C-front or C-fawn, possibly French-inspired) was introduced.
 - The language was initially called **C with Classes (CP)**.
5. **Cfront Functionality:** Similar to how TypeScript is run in Node.js, Cfront would strip out the Object Oriented principles and libraries when the code executed. It would run the C code and inject OOP features where necessary.
6. **Success:** This concept was interesting because **performance did not drop**.
7. **Final Product:** After achieving this successful integration, Stroustrup designed a new compiler, packaged everything, and named it **C++**.

C++ is still under active development with frequent versions and updates. The foundation of the language is very strong, so concepts learned remain applicable in all subsequent versions.

"There are only two kinds of languages: the ones people complain about and the ones nobody uses." - Bjarne Stroustrup

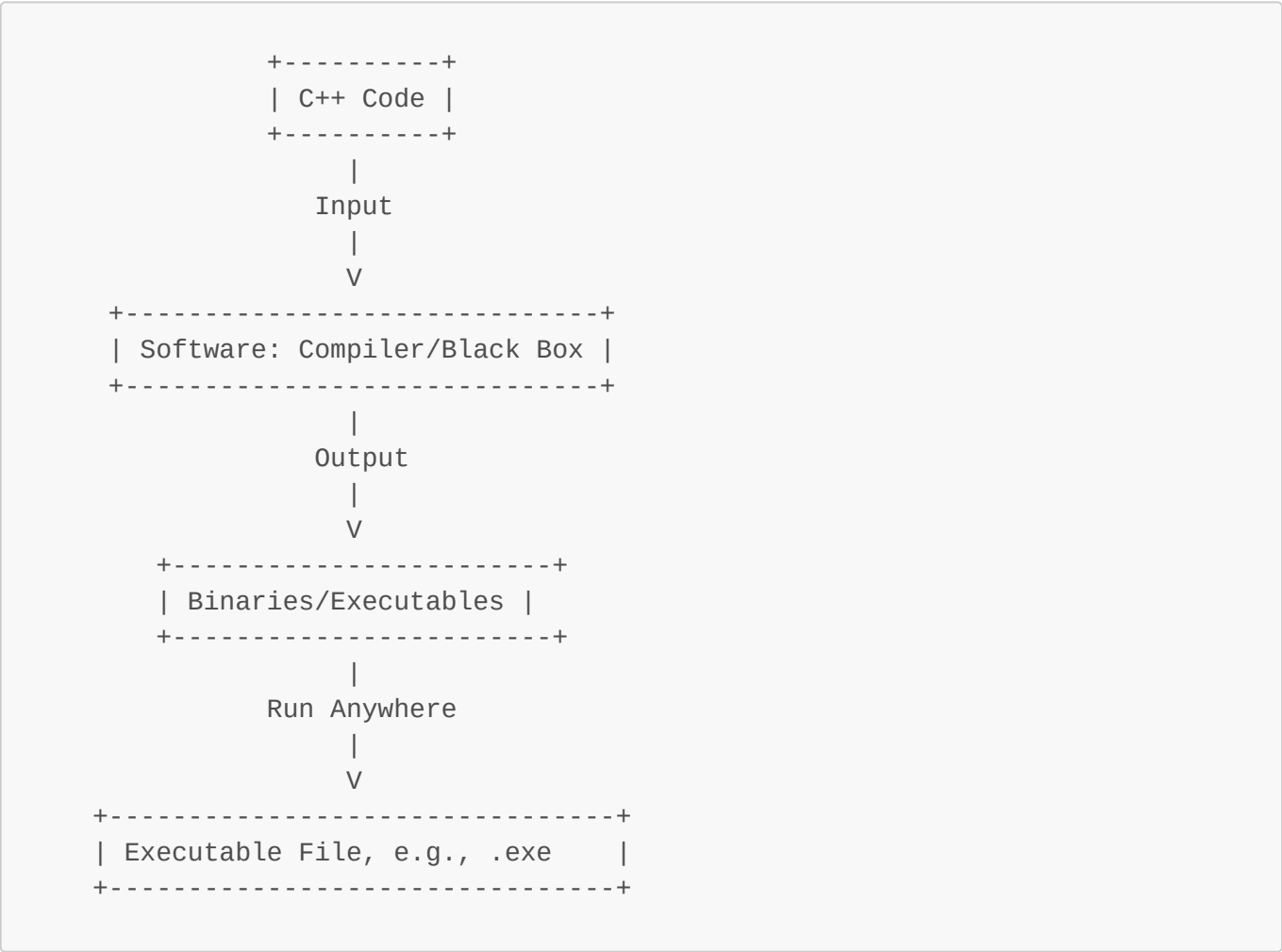
5.3. C++ Major Versions

Version	Key Features/Notes
C++11	Considered the most major version by the instructor. Introduced concepts to modernize the language, such as Lambda functions and Smart Pointers .
C++14	Introduced Generics .
C++17	Included minor updates.
C++20	Released.
C++23	Released. Showed influence from Rust in its error handling style.

6. C++ Environment Setup

6.1. Compiled Language Concept

C++ is a **compiled language**. Unlike scripting languages, C++ code requires a compiler to run.



The process is as follows:

1. The C++ code file is fed into the compiler software.
2. The compiler produces **binaries** (executables).
3. Examples of executables include **.exe** files (easy to understand).

4. This compiled approach ensures **portability** (binaries run anywhere) and **speed** (calculations are finalized beforehand).

6.2. Official Documentation

It is important to read documentation. The official C++ standard documentation is **paid**.

- **Website:** ISO CPP <https://isocpp.org/> .
- **Purchase Location:** The official standard documentation must be purchased at a National Body Store, such as an ANSI Store.
- *Note:* The website explains why working materials are free on GitHub but the standard must be purchased through ISO.
- *Microsoft C++, C, and Assembler documentation:* <https://learn.microsoft.com/en-us/cpp/?view=msvc-170>, Free.

6.3. Compilers and Tools

We require C++ binaries (compilers) to write, compile, and execute our code.

Operating System	Recommended Tooling	Core Compilers	Alternatives
Mac	Xcode (Provides everything needed for C++ development).	Clang and CMake (these are the core compilers/tools that create the application).	-
Windows	Visual Studio (Community Edition is free; Professional requires payment).	Clang and CMake .	g++ compiler ; MinGW (very famous among software engineers, can be downloaded from SourceForge).

Note: Turbo C is available but should not be downloaded.

6.4. Setting up VS Code (Step-by-Step)

The series will use **VS Code**.

1. **Folder Structure:** Bring up a VS Code instance. Create an empty folder for C++, e.g., **test**.
2. **File Creation:** Create a new file inside the folder, naming it **hello.cpp**. The extension for C++ files is **.cpp**.
3. **Install C++ Extension Pack:** Upon creation, VS Code will offer suggestions. Install the C++ Extension Pack. This pack automatically installs development environment components like **c++ theme**, **cmake**, and **cmake tools**.
4. **Install Code Runner:** Install the **Code Runner** extension (by Jun Han). This tool is highly recommended and provides the "Run Code" option.
5. **Write Code:** C++ code requires semicolons (**;**).


```
#include <iostream>

using namespace std; // Use this line, followed by a semicolon

int main() {
    // Parentheses and curly braces are required
    cout << "Hello Chai From Bali"; // Use cout (not count). The arrows
    (<<) must point toward cout to send the output

    // The semicolon must be placed outside the string

} // Semicolon must be outside the string.
```

6. **Execute Code:** Go to the drop-down menu and click **"Run Code"**. The output will appear: **Hello Chai From Hitesh.**

Note: If you need to change the compiler, you can go to settings and choose the binary (e.g., C Lang, C++, g++).

6.5. Alternative Code Environments

Environment	Pros	Cons / Restrictions
Online Compilers (e.g., Online GDB)	They are good and readily available.	Run on someone else's server, leading to restrictions. You cannot read/upload/download files. Time segmentation readings will show the standard server time, which can cause confusion. Use only if installation is impossible (e.g., company laptop).
GitHub Code Spaces	Available for C++ development.	-
Replit	Available (offers 2-3 free repls). You can create a new repl using the C++ template (e.g., name it Chai aur CPP).	The environment is acceptable, but RAM/CPU are limited, requiring more power for faster work.

7. Next Steps

The primary goal is that your code runs, regardless of the tools you use.

A separate channel has been created on Discord for C++ help.

After finishing this initial setup:

1. You should have a working code that prints "Hello Chai From...".
2. Take a selfie with the output displayed.
3. Post the selfie on Instagram and tag the channel.

The first phase (gaining interest, learning history, and installation) is complete.

The instructor will use VS Code throughout the series. Users are free to use any editor (Vim, Sublime, Xcode, online compiler). Code files will be pushed to GitHub at the end of the series.

It is highly recommended to **code along** because typing is crucial for learning. C++ will be easily understood, and we will cover more than what is strictly required.

Code

```
#include<iostream>
int main(){
    std::cout<<"Hello Chai from Bali\n";
    return 0;
}
```

Chapter 2: Anatomy of the Hello World C++ Program

1. Introduction and Context

This detailed analysis aims to fully understand the basic "Hello World" program written in C++.

Although the "Hello World" program was successfully printed in the last video, the underlying concepts were not clear. Printing "Hello World" is considered an achievement in programming.

1.1 Challenges in Programming

- Programming attracts many people due to high salaries and startup opportunities.
- The main problem is that programming requires **immense patience** (बहुत सब्र). Many people start but leave halfway.
- Even in the small program (4 or 5 lines), people likely made at least 10 mistakes, and the user might have made one or two.

1.2 Video Goals and Best Practices

- This video aims to dissect (doctor) the program and examine the mistakes made.
- These are genuine mistakes; sometimes the program executes and runs even without fixing them, but there is a significant difference between writing an efficient program and just writing something that runs.
- The goal is to understand every detail of the program and explore opportunities for optimization and improvement.
- The discussion will focus on **best practices** from Day 1 (which can also be called optimization).
- The entire program will undergo "anatomy"—a complete dissection and deep dive into every single component.
- C++ is an interesting language because it helps in understanding a lot about the system.

2. Program Execution Flow

The basic C++ program written is roughly seven lines long, but it can be converted to be shorter or longer.

```
+-----+
| helloworld.cpp | (Source Code)
|   (Input)   |
+-----+
      |
      v
+-----+
| C++ Compiler | (Converts .cpp)
+-----+
      |
      v
+-----+
|   hello   | (Executable: .exe/.out)
+-----+
      |
      v
+-----+
| OS/Terminal | (Starts at main)
+-----+
```

2.1 OS Interaction and Executables

- We can visualize the process with an Operating System (OS).
- Suppose the entire program file is named `helloworld.cpp` (or just `.cpp`).
- The `.cpp` file (source code) **does not execute directly**.
- A C++ compiler is responsible for converting the `.cpp` file into an executable format.
- This executable can be thought of as a `.exe` file (for those coming from Windows) or an executable file named `hello`.

2.2 The Starting Point: The `main` Method

- When the OS is given an executable file, it needs to know where to begin execution.
- A standard has been set: all C++ programs **must start from the `main` method**.
- This standard ensures that all OSs know that if a C++ executable is provided, it will contain a `main` method.
- The OS transfers control to this `main` method.
- Therefore, every program file must include a `main` method.

3. Anatomy of the C++ Program Lines

3.1 Preprocessor Directives: `#include <iostream>`

The first part of the study focuses on the line:

```
#include <iostream>
```

- **Definition:** Any line that starts with a hash sign (#) is called a **Preprocessor Directive**. There are many types of preprocessor directives.
- **Inclusive Directive:** The specific directive `#include` is an **Inclusive Director**.
- **Functionality:** It signifies the need to "include something" or "use something".
- It instructs the compiler to include all the code and functionality written inside the file named `iostream` into the current program.
- **Why it is Needed:** When a programmer uses `cout` inside the `main` function, the program does not automatically know what `cout` is; its definition must be written somewhere. That definition resides within the `iostream` file.
- By including this file, the program is **totally allowed to borrow** any necessary functionality from `iostream`.
- **Contents of `iostream`:** The `iostream` file controls **Input Output Streams**.
 - It allows basic operations like `cin`, `cout`, `cerr`, and `clog`.
 - It controls taking input (e.g., from the command line) and providing output (e.g., to the terminal). (Note: Other streams exist for more complex I/O, such as reading from Excel or PDF files).

3.2 Whitespace and Compiler Smartness

- Extra line spaces or whitespaces generally **do not matter**.
- C++ code starts as text format.
- The compiler processes the code through several iterations:
 1. **Syntax Check:** Checks if the syntax is correct (e.g., ensuring `include` is spelled correctly).
 2. **Token Parsing:** Identifies special words that have meaning (e.g., `#include`, `using`, `int`). This is called token parsing. (The colors seen in VS Code are based on token parsing color customization).
- During compilation, the compiler is smart and automatically removes all unnecessary extra spaces.
The code is not optimized by adding or removing extra whitespaces.

3.3 Namespaces: `using namespace std;`

The next line frequently seen is:

```
using namespace std;
```

- **The Problem:** Methods defined by the developer (e.g., `hitesh` or `chai`) might conflict with methods that already exist internally within the C++ structure (e.g., `cout`, `cin`).
- **The Solution (Namespaces):** C++ developers created separate containment areas, or "boxes," for grouping code. This box is called a **Namespace** (also referred to as a "zone" or "region" in C++).
- **Standard Namespace (`std`):** All the standard C++ code (including `cout` and `cin`) is written inside a standard box named `std`.
- **Containment:** The code written inside the standard namespace does not affect external code, and vice versa.
 - If a user wants to borrow something from the standard namespace (by `using namespace std;`), they must be mindful not to name their own methods the same (e.g., don't create a custom method named `cout`).

- **Custom Namespaces:** It is entirely possible to design and create your own namespaces.

```
+-----+
| Standard Namespace (std:: Zone/Region) |
|                                         |
| +-----+ +-----+ +-----+ |
| | cout  | | cin   | | endl  | | (Standard Functionality)
| +-----+ +-----+ +-----+ |
|                                         |
| (User code must borrow functionality) |
+-----+
```

Example of a Custom Namespace

```
namespace MyChai { // Namespace definition using curly braces
    // ...
    void Display(); // Method definition
    // ...
}
```

The method inside the namespace can be called using the namespace name: `MyChai::Display()`.

Real-World Examples of Namespaces

Namespaces are common in C++ frameworks and libraries:

- **Qt Framework:** Provides namespaces for designing widgets and simplifying APIs <https://www.qt.io/product/framework> .
- **Eigen:** A famous, highly active C++ library used primarily for mathematics. It involves linear algebra, matrices, and vectors, and is used extensively in machine learning <https://github.com/PX4/eigen> .
- **GTest:** Another famous framework/namespace bu google for testing <https://github.com/google/googletest> .

3.4 Methods for Using Namespaces

While `using namespace std;` loads the entire namespace, which is generally considered an acceptable practice because C++ optimizes it, there are alternatives.

Method 1: Using the Scope Resolution Operator (::)

The full `using namespace std;` line can be removed. Instead, specific elements needed from the Standard namespace are prefixed with `std::`.

Standard Notation:

```
std::cout << "Hello World";  
std::endl; // For example, the end line operator
```

- The two colons (::) indicate that the method being called belongs to the `std` namespace.
- This method means the programmer is "nit-picking" specific methods from the namespace.

Method 2: Importing Specific Elements with `using`

This pattern is often seen in open-source C++ code. Specific elements are imported using the `using` keyword:

```
using std::cout;  
using std::endl;  
// Now cout and endl can be used without the std:: prefix
```

All three methods (full load, `std::` prefix, or specific imports) achieve the same fundamental goal.

3.5 The `main` Function and Return Types

Functions and Functionality

- Methods are also called **Functions**.
- A function's purpose is to bring functionality. For example, the `+` function adds numbers on its left and right sides.
- The `main` function's specific job is to **start the program**.
- Every function must have a defined functionality.

Rules for Function Definition

1. **Naming:** Meaningful names should be used (e.g., `getAPIFromGitHub`), avoiding ambiguous names like `hitesh` or `chai` (except for `main`).
2. **Static Typing and Return Type:** C++ is a **statically typed language**. This means the data type (number, string, etc.) must be specified beforehand. This rule applies to functions.
 - A function must declare **what type of value it will return** when it finishes execution.
 - In the standard `main` definition, we use `int` (Integer, referring to whole numbers like 1, 2, 3, 4, 5).

The Perfect, Valid, and Smallest Function:

```
int main() {  
    // Function body  
    return 0;  
}
```

- Since `int` (integer) is defined as the return type, the function must explicitly return a value of that type.

Return Values and Exit Codes

- You can return any integer, such as `return 5;`.
- C++ uses **Exit Codes** to define the result of function execution.
 - If `return 5` is used, the code will exit "with code 5".
- The **most common Exit Code is 0**.
- `return 0` means the function exited successfully and did not need to return any specific data.
- Returning 0 provides predictability if the program runs in other environments.

3.6 Output Operators and Semicolons

- The symbols used for passing values to `cout` (e.g., `<<`, referred to as two less-than signs or greater-than signs) are operators, similar to the `+` operator.
- **Operator Function:** These operators take the value (e.g., a string) from the right side and **pass it on** to the left side.
- `endl`: The `endl` instruction also gets passed on. `endl` represents a line end (the equivalent of pressing the 'Enter' key).

Semicolon Rule

- In C++, generally, **every line must end with a semicolon** `;`.
- Semicolons are not strictly required in certain places (like after preprocessor directives or function headers), but they are necessary for ending statements.

4. Summary of Key Concepts

A quick summary of the concepts covered:

Concept	Detail
<code>#include</code>	Used for including functionality (e.g., <code>iostream</code>). It is a Preprocessor Directive.
Namespaces	Used to organize and contain code. Can be included fully (<code>using namespace std;</code>), partially (<code>using std::cout;</code>), or explicitly on the go (<code>std::cout</code>).
<code>main</code> Method	Every C++ file requires a <code>main</code> method as the program entry point.
Static Typing	The C++ language requires data types to be defined explicitly.
Return Type	The return type of <code>main</code> (usually <code>int</code>) must be defined.
<code>cout</code>	The output functionality, whose role and origin are defined in <code>iostream</code> .
Output Operators	Operators (<code><<</code>) pass string or value data from right to left.

Concept	Detail
Exit Code	<code>return 0</code> is the standard exit code indicating successful execution. <code>return 5</code> is possible, but <code>0</code> is the guideline for success.

This detailed study of "Hello World" provides a basic foundation, touching upon concepts related to the Operating System and Compiler Design (which is an engineering subject itself).

Code

```
#include<iostream>
// #include : Preprocessor Directive, used to include something
// <iostream> i/o operations lib, cin, cout, cerr, clog

// whitespaces/ extra lines dont matter in cpp
// compiler does syntax check & token parsing
// code colours in vsc = token based

using namespace std;
// import lib stuff way 1
// using the region/namespaces directly in code, no need to write
region::func
// std region/namespaces contains stuff , internal cpp code, like cin,cout
etc
// helps to separate intermixing of cpp internal region & our code
project's region

// using std::cout;
// using std::endl;
// // import lib stuff way 2
// // import specific element using using

namespace bali{
    void display(){
        cout<<"Simply Lovely"<<endl;
        // < , this operator takes value at right and passed on the left
        side, similar to + operator
        // endl represents line end, like pressing enter key
    }
}
// custom namespace
// if making custom namespace, avoid writing c++ internal keywords, or just
use it as custom::func

int main()
// main func is the starting function of program as per decided by cpp
standards
// func purposes is to bring methods, meaningful work
// as cpp is static typed lang, not dynamic, we have to explicit declare
every datatype during objects creation
// int is stated with main() func
```



```
// if no return type , then state void datatype
{
    bali::display();// useage of custom namespace
    // std::cout<<""; // import lib stuff way 3

    return 0;
    // cpp uses exit codes to define result of func execution
    // most common 0 : successful exit, and no need to return any spec data
    // 0 provides common predictability in other environments
    // return 5;
    // // exit codes can be customised based on projects standards and env
    standards

}
```

Chapter 3: Variables and Constants

I. Introduction and Series Context

This content is part of the C++ series titled "**Chai aur Code**," which is also referred to as "**Chai aur C++**". The series is widely followed, noted by the phrase "pura India yahin se padh raha hai" (the whole of India is reading/learning from here nowadays).

Video Goals and Pacing

The video is intentionally kept at a **slow pace** so that the audience can enjoy the content. The video covers a limited number of topics because it focuses on the starting phase of programming.

Note on Starting Programming: In the beginning phase of programming, anxiety and lack of understanding are common feelings. The goal is to understand the basics, C++, and perform some practice.

II. Setting Up the Code Structure

File and Folder Naming

We start with a fresh video and a fresh start, even if some initial work was covered previously.

1. **Folder Structure:** The folder structure starts with **01** for easy organization.
2. **Best Practices:** All files and structures are created using best practices.
3. **Folder Name:** The folder is named **01_var_const** because the focus is on **variables and constants**.
4. **File Name:** The file created inside the folder is typically named **main.cpp**.

Why **main.cpp**?

Using **main.cpp** is a very common syntax. When you create **hello.cpp**, it usually includes a main method.

In large software development, a single folder might contain many supporting files (**.h** files and **.cpp** files). Since the end goal is to create one executable software, **main.cpp** is often the name given to the primary

execution file, especially when it is the only executable file or when it is unclear which file among thousands should be run.

III. Basic C++ Program Structure

The following code is written as an exercise and revision.

Component	Description/Function	Revision Questions (Asked in Source)
<code>#include</code>	Used to include libraries.	What is <code>#include</code> ?
<code>iostream</code>	Input/Output stream.	What is <code>iostream</code> ?
<code>using namespace std;</code>	Refers to a container (like the box example previously discussed). Semicolon is very necessary here.	
<code>int main()</code>	The basic method structure.	
<code>cout</code>	Responsible for logging any given string or value to the console (standard output).	
<code>endl</code>	Stands for End of Line . It ends the current line. Note: This is one of several mechanisms available to end a line.	
<code>return 0;</code>	A sign of a good program. It is required because the method declared that it will return an integer (<code>int main</code>). Semicolon must not be forgotten.	

Code Block: Basic C++ Program

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

int main() {
    // Basic output structure
    cout << "Welcome to Chai with CPP" << endl;

    // Good practice
    return 0;
}
```

IV. Understanding and Using Comments

Comments are often seen inside programs.

Purpose of Comments

The primary use of comments is when you have many lines of code or many methods and you need to determine which part is causing a problem. You use comments to temporarily remove (hide) certain code lines and test the program.

Compiler Interaction with Comments

The compilation process involves the **Code File** (a basic text file) going through a **Compiler** (the C++ compiler) to produce an **Executable**.

1. During compilation, the compiler extracts and gathers all keywords (like `int`, `main`, `namespace`) to build an Abstract Syntax Tree (AST).
2. When a line is commented out, the **compiler ignores that line totally**.

Types of Comments and Shortcuts

Type	Syntax/Method	Usage
Single Line Comment	Press Control + Slash (/) .	This is an almost universal shortcut in VS Code, regardless of the programming language (C++, Java, JavaScript). It is used to toggle commenting on and off. This is the professional developer preference.
Multi-Line Comment	Start with <code>/*</code> and end with <code>*/</code> .	This method exists but is often favored by academic standards.

Professional Commenting Best Practices

While academics often suggest writing a comment before every line or method:

- **Avoid comments that add no value** (e.g., "This line prints a welcome message"), as the code itself is clear.
- It is considered a **bad practice** in real-world software development to write unnecessary comments.
- **Comments should only be written where complexity has been introduced**.

White Spaces

Extra white spaces (e.g., excessive line breaks or tabs) do not affect the compiler. The compiler has **no concern** with white spaces.

VS Code Line Manipulation

A useful shortcut for moving code lines is pressing **Option** (or **Alt**) and the **Down Arrow** key. This is easier and faster than cutting and pasting code.

V. The Fundamental Role of Data

The main reason for creating any program or software is **data**. Data-related operations generally fall into three categories:

1. **Store:** Storing data.
2. **Process:** Manipulating data.
3. **Display/Read:** Showing the result to the user (the ultimate goal of data).

Data is typically displayed only **after** it has been stored or processed.

Data Storage

Storage is necessary because data needs to be maintained (e.g., maintaining a user's score or username in a game or a bank balance). Data can be stored in memory (RAM), a disk, or a database.

- **Concepts related to Storage:** Variables, Constants, Data Types (Integers, Strings, Booleans/True-False), Data Structures (Arrays, Objects, Link Lists, Custom structures).

Data Processing

Processing involves taking stored data and acting upon it.

- **Example:** Checking if a stored username contains only numbers and characters, or if it contains unauthorized special characters.
- **Concepts related to Processing:** Conditional statements (**if-else**), Loops, **switch** cases, and Object-Oriented Programming (OOP).

VI. Variable and Constant Declaration

We need to store data using variables, rather than just repeatedly using **cout**.

Variables

Variables are like a box or a chair that can be **opened and changed**. Data inside a variable can be modified.

1. Declaration (The Syntax)

When you declare a variable, you are asking the RAM (Random Access Memory) for a small box and giving that box a name (an identifier).

```
int score;
```

- **int:** Specifies the data type (in this case, an Integer) that will be stored.
- **score:** The name given to the memory location (the box).
- **Vocabulary:** This specific step (**int score;**) is called **Variable Declaration**.

2. Initialization (Assigning Value)

Initialization means assigning a value to the declared variable.

Two-Step Process:

1. Declaration:

```
int score; // Declared, but value unknown/unassigned
```

2. Initialization/Usage:

```
score = 110; // Value assigned later
```

Combined Declaration and Initialization (Single Step):

If the value is known from the beginning, you can combine these steps.

```
int balance = 500;
```

Why Use Two Steps?

Sometimes, you need to reference the memory location first, and then add the value after some processing occurs or data is read from a database. Programming always provides options.

3. Modification

Once a variable is declared, you use it directly without re-declaring it. Changing the value is allowed:

```
int hiteshBalance = 500;  
hiteshBalance = 1000; // Change is complete and totally allowed
```

Identifiers (Variable Naming Rules)

The names given to variables are called **Identifiers**.

Keywords

You cannot use the language's reserved **keywords** (special words like `int` or `for`) as variable names. The compiler will give errors.

Allowed Naming Conventions

Variable names should be **meaningful**. Various styles are allowed and common:

1. **Camel Case:** `hiteshBalance` (using capitalization).
2. **Snake Case:** `hitesh_balance` (using underscores).

- Underscores are sometimes used to imply that a variable holds a special meaning or should not be exposed externally.

3. Generalized: `balance`.

- ### 4. Numbers at the End:
- Names like `hiteshBalance1` or `hiteshBalance2` are allowed, although the speaker generally prefers meaningful names without trailing numbers.

What is NOT Allowed

1. **Spaces:** Spaces are **not allowed** (e.g., `hitesh balance`) because the compiler treats them as separate variables.
2. **Starting with an Integer:** Names that start with a number or are structured confusingly (e.g., if `one` is used in place of a variable name).

Constants

Constants use a special keyword to ensure the data stored **cannot be changed**. This is useful for values that must remain fixed throughout the program (e.g., a unique customer ID).

1. Declaration using `const`

The `const` keyword is used before the data type:

```
const int uniqueID = 232323; // This uniqueID is now constant
```

2. Attempting to Modify a Constant

If you try to change the value of a constant, the compiler generates an error.

```
// Attempting to change uniqueID  
uniqueID = 121212;
```

Error Message: The attempt will result in an error stating that the **expression must be a modifiable lvalue**.

- **L-Value:** Left-Hand Side value.
- **R-Value:** Right-Hand Side value.

The compiler indicates that the Left-Hand Side value (`uniqueID`) cannot be changed because it was declared with `const`.

Left-Hand Side value & Right-Hand Side value

L-value (The Mailbox)

An **L-value** refers to a **memory location** that has an address. Because it's a location, you can go back to it later to store new things or see what's there. It's the mailbox itself.

- **Key Property:** It has a persistent address. It's a **container**.
- **What you can do:** You can put something **into** it (assign a value to it) and you can also see **what's inside** it (read its value).
- **Mnemonic:** Think **L** for **L**ocation.

In code, a variable is a perfect example of an L-value.

```
int score = 95;
// 'score' is an L-value. It refers to a specific spot in memory
// where the number 95 is stored.

score = 100; // This is VALID because you can put a new value into the
'score' location.
```

R-value (The Letter) ✉

An **R-value** is a **temporary value** that doesn't have a persistent memory location. It's the letter you are about to put *in* the mailbox. Once it's delivered (or the expression is evaluated), the letter itself is gone; only its value was used.

- **Key Property:** It does not have a persistent address. It's the **content**.
- **What you can do:** You can only **read** its value. You can't assign a new value *to* it.
- **Mnemonic:** Think **R** for **R**ead or **R**ight-hand side.

Literal numbers or the result of a calculation are common R-values.

```
int score = 95;
// '95' is an R-value. It's a pure value.

int finalScore = score + 5;
// 'score + 5' is an R-value. The computer calculates 100,
// and this temporary result of 100 is the R-value.
```

The Golden Rule: Assignment

The names "Left-Hand Side" and "Right-Hand Side" come from the assignment operator (=).

An L-value can be on the left or right side of =, but an R-value can ONLY be on the right side.

Let's see why:

```
int x; // x is an L-value (a memory location/mailbox)

x = 10; // Correct!
        // L-value = R-value
```

```
        // (Put the letter '10' into the mailbox 'x')

10 = x;    // ERROR!
           // R-value = L-value
           // (This is like trying to assign the mailbox to the letter. It
           // makes no sense!)
```

Why This Mattered in Your Last Example

You saw the error `expression must be a modifiable lvalue`. Let's break that down with what we know now:

```
const int UNIQUE_ID = 12345;
UNIQUE_ID = 54321; // ERROR!
```

- `UNIQUE_ID` is an **L-value** because it has a memory location.
- However, because you marked it `const`, the compiler made it a **non-modifiable L-value**.
- The error is telling you: "The thing on the left-hand side (`UNIQUE_ID`) is a location, but I am not allowed to modify it!"

Feature	L-value (Location)	R-value (Value)
Analogy	Mailbox 📬	Letter ✉️
Has Address?	Yes, a persistent memory location.	No, it's temporary.
Can be on LHS of <code>=</code> ?	Yes (if not <code>const</code>).	Never.
Can be on RHS of <code>=</code> ?	Yes.	Yes.
Example	<code>int x</code> ; (<code>x</code> is the L-value)	<code>100</code> , (<code>x + 5</code>)

VII. Next Steps in Programming

The next step is to master data storage concepts by understanding other available data types:

- How to store **String** data.
- How to store **Boolean** (True/False) values.

Once data storage is fully understood, the focus shifts to **data processing** (how data can be processed in different ways). The full path involves understanding storage, then processing, and then building logic and programs.

Of course! Let's break down L-values and R-values with a simple analogy. This concept is fundamental to understanding how C++ handles data and memory.

Think of your computer's memory as a street lined with mailboxes.

Code


```
#include<bits/stdc++.h>
// main.cpp is a very common syntax. In large software development, main.cpp
is often the name given to the primary execution file
using namespace std;
int main(){
    // Comments
    // The compilation process involves the Code File (a basic text file)
going through a Compiler (the C++ compiler) to produce an Executable.
    // 1. During compilation, the compiler extracts and gathers all
keywords (like int, main, namespace) to build an Abstract Syntax Tree
(AST).
    // 2. When a line is commented out, the compiler ignores that line
totally.

    // ctrl + / VSC Shortcut
    /*
Multiline comment
*/

    // Avoid comments that add no value, It is considered a bad practice in
real-world software development to write unnecessary comments.
    // Comments should only be written where complexity has been
introduced.

    // Data
    // Store: Storing data. DS
    // Process: Manipulating data. A
    // Display/Read: Showing the result to the user (the ultimate goal of
data).

    // Variables
    // When you declare a variable, you are asking the RAM (Random Access
Memory) for a small box and giving that box a name (an identifier)
    // Declaration
    int score                ;    // Extra white spaces (e.g.,
excessive line breaks or tabs) do not affect the compiler. The compiler has
no concern with white spaces
    // Initialization/Usage
    score = 100;
    // Combined Declaration and Initialization
    int balance =1;
    // Modification
    balance = 999999; // Change is complete and totally allowed

    // Identifiers (Variable Naming Rules)
    // NO Keywords
    // YES baliking // camelCase
    // YES bali_king // snake_case
    // NO 1bali // no start with int
    // NO bali king // no spaces
    // YES _baliKey // typical norm to show sensitive info
```

```
// Contants
// if you mark any variable const, the compiler made it a non-
modifiable L-value.
// If you try to change the value of a constant, the compiler generates
an error.
const int rollno1 = 122;
// rollno1 = 0; // NO, ERROR
// // error: assignment of read-only variable 'rollno1'

// Left-Hand Side value
// where variable is at left
// persistent
int x = 10; // LHS => x
// Right-Hand Side value
// var also at right
// inpersistent
int y = x*x ; // RHS => x*x

// NO int y = (x**2);
// C++ does not use ** for exponentiation (calculating powers).
// use std::pow(number, 2)

return 0;
}
```

End-of-File

The [kintsugi-stack](#) repository, authored by Kintsugi-Programmer, is less a comprehensive resource and more an Artifact of Continuous Research and Deep Inquiry into Computer Science and Software Engineering. It serves as a transparent ledger of the author's relentless pursuit of mastery, from the foundational algorithms to modern full-stack implementation.

Made with  [Kintsugi-Programmer](#)