Writing Our First C++ Program & Basic Structure:

1.**Output (`cout`):** To display text on the screen, you use the `cout` statement. You write `cout <<` followed by the text you want to print inside double quotes (`"`). This is called an output statement.

2.**Case Sensitivity:** C++ is a **case-sensitive language**. This means `cout` (lowercase) is different from `Cout` or `COUT`. You must use the correct case as shown in the code. Languages like Java and Python are also case-sensitive, while SQL and HTML are not.

3.**Semicolons (`;`):** Every statement in C++ must end with a **semicolon**. The semicolon acts as a **statement terminator**, similar to a full stop or period in English.

4.**Main Function (`int main()`):** Every C++ program requires a `main` function. This function is the **starting point of execution** for the program, analogous to the "Start" block in a flowchart. Code execution always begins here. The main function is written as `int main()` followed by parentheses `()` and then curly braces `{}`. The code you want to execute goes inside the curly braces. Don't worry too much about the details of functions now, as they will be covered later.

5.**Include Directive (`#include <iostream>`):** At the top of the file, you need to include `#include <iostream>`. This is a **preprocessor directive** that tells the compiler to include the `iostream` file, which contains the logic for input/output operations like `cout`. Without including this file, the compiler wouldn't know how `cout` works, leading to errors. This line should be included in every C++ code file.

6.**Namespace (`using namespace std;`):** Another line typically written at the top is `using namespace std;`. A namespace can be imagined as a file that contains the logic for various things, like `cout` or `cin`. The `std` namespace is C++'s standard namespace. By including this line, you tell the compiler to use the `std` namespace throughout the code. This avoids having to write `std::cout` or `std::cin` every time you use them. While this line is optional, if omitted, you must prefix `cout`, `cin`, and `endl` with `std::`.

7.**Return Statement (`return 0;`):** Inside the `main` function, before the closing curly brace, you should write `return 0;`. Although the code might run without it, including `return 0;` is considered a good programming practice, as the `int main()` function is designed to return an integer value (0 typically signifies successful execution).

6.**Running Code (Two Stages):** C++ code runs in two stages:

◦**Compilation:** The compiler checks the code for syntax errors and translates it into an executable file (machine code) that the computer understands (0s and 1s). To compile, type `g++ <filename>.cpp` (e.g., `g++ code.cpp`) in the terminal and press Enter. If successful, an executable file is created.

◦**Execution:** The executable file is run by the computer. On Windows, the executable is typically named `a.exe` or something ending in `.exe`. On Mac/Linux, it's typically named `a.out`. To run the executable, type `./a.out` (for Mac/Linux) or `a.exe` or `.\a.exe` (for Windows) in the terminal and press Enter.

7.**Combined Compile and Run:** You can combine these two steps using the `&&` operator: `g++ <filename>.cpp && ./a.out` (or the Windows equivalent). This first compiles the code and then runs it if compilation is successful.

8.**Clearing Terminal:** Use the `clear` command in the terminal to clear the screen.

9.**Recalling Commands:** Use the up/down arrow keys on the keyboard to cycle through previous commands entered in the terminal.

Output Formatting (Newline Characters):
•By default, `cout` prints everything on the same line. After printing, a strange percentage-like sign might appear. This happens because the output doesn't end with a newline character.
•To move to the next line after printing, you can use either `endl` or `\n`.
•**`endl`:** Append `<< endl;` after the output you want to terminate with a newline. This is commonly used and easy to type.
•**`\n`:** Use `\n` inside the double quotes of the `cout` statement where you want the newline to occur. `\n` is generally slightly faster than `endl`, which might be relevant in competitive programming, but for learning purposes, both work.
•You can concatenate multiple outputs and newline characters using the `<<` operator.

Boilerplate Code and Comments:

•The initial lines required for every C++ program (`#include <iostream>`, `using namespace std;`, `int main() { ... return 0; }`) are referred to as **boilerplate code**. This is the basic structure for almost any C++ program you write.

•**Comments:** Comments are notes written in the code that are **ignored by the compiler**. They are meant for humans (other developers or your future self) to understand the code. Comments do not follow programming rules.

•**Single-line comments:** Start a line or a part of a line with `//`. Anything after `//` on that line is a comment.

•**Multi-line comments:** Not explicitly covered in the excerpt, but typically enclosed within `/* ... */`. (Self-correction: The source only shows single-line comments with `//`. Stick to the source).
•In VS Code, comments are typically displayed in green.

•**Shortcut for commenting:** `Ctrl + /` (or `Cmd + /` on Mac) can comment out or uncomment selected lines.

Variables:

•**Concept:** Variables are like containers in programming that store data. They are similar to variables used in mathematics (e.g., `a = 5`). You give a variable a name (e.g., `age`, `price`, `grade`) and store a value in it.

•**Memory Storage:** When you define a variable (e.g., `int age = 25;`), a specific location in the computer's memory (RAM) is allocated. This memory location is given the variable's name, and the value is stored there. The value inside the variable can change over time, hence the term "variable".

•**Declaration:** To create a variable, you need to specify its **data type** and its **name**, optionally assigning an initial value. The data type must be declared before the variable name (e.g., `int age = 25;`).

•**Naming Rules (Identifiers):** Variable names are also called **identifiers**. They must start with either an underscore (_) or an English letter (a-z, A-Z). They cannot start with a digit (0-9). Names are typically in English.

•**Printing Variable Values:** To print the *value* stored in a variable, use `cout << <variable_name>;` **without** double quotes around the variable name. If you put double quotes, the variable name itself will be printed as text.

Data Types:

•A data type tells the compiler **what kind of data** a variable will hold and **how much memory** it should allocate for it.

•**Integer (`int`):** Used for storing **whole numbers** (positive, negative, or zero) that do not have a decimal point. Internally, an `int` typically occupies **4 bytes** of memory.

•**Character (`char`):** Used for storing a **single character** (like 'a', 'B', '5', '$'). Characters are enclosed in **single quotes** (`'`). A `char` typically occupies **1 byte** of memory.

•**Float (`float`):** Used for storing **floating-point numbers** (numbers with a decimal point, e.g., 3.14, 1.99)15. These are also called decimal values. A `float` typically occupies **4 bytes** of memory. When assigning a literal value to a `float`, it's good practice to append `f` or `F` (e.g., `3.14f`) to tell the compiler it's a float, otherwise it might assume it's a `double` by default.

•**Boolean (`bool`):** Used for storing **Boolean values**, which can only be either `true` or `false`. A `bool` typically occupies **1 byte** of memory. When printed, `true` often translates to `1` and `false` to `0` in the output, as this is how they are represented internally.

•**Double (`double`):** Used for storing **floating-point numbers** with **higher precision** than `float`. A `double` typically occupies **8 bytes** of memory (double the size of a float). Decimal numbers written as literals (e.g., `1.99`, `100.99`) are treated as `double` by default.

•**Primitive Data Types:** `int`, `char`, `float`, `bool`, and `double` are considered **primitive data types** – the most basic data types in C++. Other data types like arrays and strings (which will be covered later) are non-primitive.

Memory Representation & Size:
•Computers work with the **binary number system** (0s and 1s).
•A single binary digit (0 or 1) is called a **bit**.
•A collection of eight bits is called a **byte**.
•The size of data types (e.g., `int` is 4 bytes) determines how many bits are allocated in memory to store a value of that type. A 4-byte integer occupies 32 bits (4 * 8).
•Values are stored in memory in their binary form. For example, the integer 25 is stored as `11001` in binary.

•**ASCII Values:** For `char` data types, single characters are stored in memory based on their **ASCII (American Standard Code for Information Interchange) value**. Each character has a standard numerical representation (e.g., 'A' is 65, 'B' is 66, 'a' is 97, 'b' is 98). The ASCII value is converted to binary and stored14. This will be explored further in a dedicated chapter on strings.

•**`sizeof()` Operator:** The `sizeof()` operator is a function that returns the size (in bytes) of a data type or variable. You can use `cout << sizeof(<data_type_or_variable_name>);` to see the size.

Type Casting:

•**Concept:** Type casting (or type conversion) is the process of **converting data from one data type to another**.

•**Implicit Type Conversion (Type Conversion):** This conversion happens **automatically** by the compiler without requiring explicit action from the programmer. It usually occurs when converting a value from a **smaller data type to a larger data type**, where there is no data loss.
◦**Example**: Assigning a `char` value (1 byte) to an `int` variable (4 bytes). The character's ASCII value is converted and stored as an integer. For instance, assigning 'A' to an `int` variable will store 65.

•**Explicit Type Casting (Type Casting):** This conversion is performed **manually by the programmer**. It is often used when converting a value from a **larger data type to a smaller data type**, which **may result in data loss**.
◦**Syntax:** You put the desired new data type in parentheses before the value or variable you want to convert (e.g., `(int)price`).
◦**Example:** Converting a `double` value to an `int`. When casting a floating-point number (like 100.99) to an integer, the decimal part is simply **truncated** (cut off), not rounded. So, 100.99 becomes 100 when cast to an `int`.

Input (`cin`):
•**Concept:** To take input from the user (e.g., from the keyboard), you use the `cin` statement.
•**Syntax:** You write `cin >> <variable_name>;`. The data entered by the user will be stored in the specified variable.
•**Example:** You can prompt the user with `cout << "Enter the age: ";` and then read their input into an integer variable `age` using `cin >> age;`.

•**Garbage Value:** If a variable is declared but not assigned an initial value or hasn't received input, it contains a "garbage value" – some random value that happens to be in that memory location.

Operators:
•**Concept:** Operators perform operations on values or variables. They are similar to mathematical operators.

•**Arithmetic Operators:** Perform mathematical calculations21.
◦+ (Addition)
◦− (Subtraction)
◦ * (Multiplication - uses the asterisk symbol)
◦/ (Division - uses the slash symbol)
◦% (Modulo - returns the remainder of a division)

•**Integer Division:** When dividing two integers in C++, the result is also an integer. The decimal part of the result is truncated. For example, `5 / 2` results in `2`, not `2.5`. To get a floating-point result, at least one of the operands must be a floating-point type (`float` or `double`), or you can use type casting.

•**Relational Operators:** Used to compare values and determine the relationship between them (e.g., whether one is less than, greater than, or equal to another). They always return a **boolean value** (`true` or `false`, which often prints as 1 or 0).
◦< (Less than)
◦<= (Less than or equal to)
◦> (Greater than)
◦>= (Greater than or equal to)
◦== (Equal to - uses double equals sign) - Single equals = is used for assignment.
◦!= (Not equal to - uses exclamation mark and equals sign)

•**Logical Operators:** Used to combine or modify boolean expressions.
◦`||` (Logical OR - double pipe symbol): Returns `true` if **at least one** of the expressions is `true`. Returns `false` only if both expressions are `false`.
◦`&&` (Logical AND - double ampersand symbol): Returns `true` only if **both** expressions are `true`. Returns `false` if at least one expression is `false`.

◦`!` (Logical NOT - single exclamation mark): Reverses the boolean value of an expression. If an expression is `true`, `!expression` is `false`, and vice versa.
◦Logical OR and AND will be more clearly understood when covering conditional statements.

•**Bitwise Operators:** Bitwise operators are used to perform operations at the bit level in programming. They work directly on binary representations of numbers, making them efficient for tasks like optimization, encryption, and low-level system programming.

•**Unary Operators:** Operate on a **single operand** (variable).
◦`++` (Increment): Increases the value of a variable by 1 (`a = a + 1`).
▪**Post-increment (`a++`):** The variable's value is **used first**, and then it is **incremented**.
▪**Pre-increment (`++a`):** The variable is **incremented first**, and then its **updated value is used.**

◦ `--` (Decrement): Decreases the value of a variable by 1 (`a = a - 1`).
▪**Post-decrement (`a--`):** The variable's value is **used first**, and then it is **decremented**.
▪**Pre-decrement (`--a`):** The variable is **decremented first**, and then its **updated value is used.**

◦These operators are particularly useful in loops.