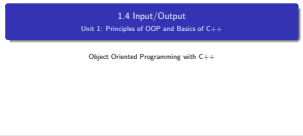
# 1.4 Input/Output

## Unit 1: Principles of OOP and Basics of C++

Object Oriented Programming with C++







# 1.4 Input/Output in C++

- Course: Object Oriented Programming with C++
- Unit 1: Principles of OOP and Basics of C++
- Instructor: Expert C++ Instructor
- Topics: Streams, cin, cout, Manipulators, Namespaces





1.4 Input/Output in C++
- Course: Object Oriented Programming with C++
- Unit 1: Principles of OOP and Basics of C++
- Instructor: Expert C++ Instructor
- Topics: Streams, cin, cout, Manipulators, Namespaces

Welcome to Lecture 4. Today, we transition from defining variables and data types in memory to interacting with the user. We will cover the standard ways C++ handles input and output, which is fundamentally different and much safer than C's printf and scanf. We'll also discuss how C++ organizes its standard library using namespaces, and how we can make our output look professional using manipulators.

# The Concept of Streams in C++

- C++ does not have built-in I/O statements.
- I/O is handled by the Standard Library via the ¡iostream¿ header.
- Stream: An abstraction representing a flow of data.
- Input Stream: Flow of bytes from a device (keyboard, file, network) to main memory.
- Output Stream: Flow of bytes from main memory to a device (screen, printer, file).





The Concept of Streams in C++
- C++ does not have built-in I/O statements.
- I/O is handled by the Standard Library via the ¡iostream¿ header.
- Stream: An abstraction representing a flow of data.
- Input Stream: Flow of bytes from a device (keyboard, file, network) to main memory.
- Output Stream: Flow of bytes from main memory to a device (screen, printer, file).

Unlike some languages, C++ core language doesn't know how to print text. It relies on the standard library. The fundamental concept here is a 'stream'. Imagine a stream of water; in C++, it's a stream of bytes. When you type on your keyboard, bytes flow into a stream that your program can read. When you want to display text, you push bytes into an output stream connected to the console. This abstraction is powerful because the same stream operations can be used for files or network sockets later on.

# Standard I/O Stream Objects

- Defined in the ¡iostream¿ library.
- cout (Console Output): Standard output stream, usually directed to the monitor.
- cin (Console Input): Standard input stream, usually attached to the keyboard.
- cerr (Console Error): Unbuffered standard error stream.
- clog (Console Log): Buffered standard error stream.





Standard I/O Stream Objects
- Defined in the ¡iostream¿ library.
- cout (Console Output): Standard output stream, usually directed to the monitor.
- cin (Console Input): Standard input stream, usually attached to the keyboard.
- cerr (Console Error): Unbuffered standard error stream.
- clog (Console Log): Buffered standard error stream.

When you include ¡iostream¿, C++ automatically creates four stream objects for you. The two we will use most are cout and cin. 'c' stands for character, so 'character output' and 'character input'. We also have cerr and clog for error handling. cerr is unbuffered, meaning errors appear immediately, which is crucial if the program crashes. clog is buffered, better for general logging that doesn't need immediate display.

# Output with cout and ¡¡

- The cout object is used with the stream insertion operator (¡¡).
- It 'inserts' data into the output stream.
- Syntax: cout ¡¡ "Text to print";
- Can print literals, variables, and expressions.
- Example: int age = 20; cout ¡¡ "I am " ¡¡ age ¡¡ " years old.";





Output with cout and ¡¡
- The cout object is used with the stream insertion operator (¡¡).
- It 'inserts' data into the output stream.
- Syntax: cout ¡¡ "Text to print";
- Can print literals, variables, and expressions.
- Example: int age = 20; cout ¡¡ "I am " ¡¡ age ¡¡ " years old.";

The double less-than sign ¡¡ is the stream insertion operator. Think of it as an arrow pointing the data into cout. You can chain these operators together to print multiple items in a single statement, which is called cascading. It automatically knows how to print strings, integers, floats, etc., because C++ operators can be overloaded to handle different data types.

# Input with cin and ¿¿

- The cin object is used with the stream extraction operator (¿¿).
- It 'extracts' data from the input stream and stores it in a variable.
- Syntax: cin ¿¿ variableName;
- Automatically determines data type based on the variable.
- Example: int age; cout ¡¡ "Enter age: "; cin ¿¿ age;





Input with cin and ¿¿
- The cin object is used with the stream extraction operator (¿¿).
- It 'extracts' data from the input stream and stores it in a variable.
- Syntax: cin ¿¿ variableName;
- Automatically determines data type based on the variable.
- Example: int age; cout ¡¡ "Enter age: "; cin ¿¿ age;

Conversely, the double greater-than sign ¿¿ is the stream extraction operator. It extracts data from cin and puts it into your variable. It skips leading whitespace (spaces, tabs, newlines) automatically. One very important rule: you must declare the variable before you can read data into it. Notice how the arrows point towards the variable, indicating the flow of data.

# Cascading I/O Operators

- Both ¡¡ and ¿¿ return a reference to their stream object.
- This allows chaining (cascading) multiple operations.
- Output Cascade: cout ¡¡ "X: " ¡¡ x ¡¡ ", Y: " ¡¡ y;
- Input Cascade: cin ¿¿ x ¿¿ y;
- Execution is from left to right.





Cascading I/O Operators
- Both ¡¡ and ¿¿ return a reference to their stream object.
- This allows chaining (cascading) multiple operations.
- Output Cascade: cout ¡¡ "X: " ¡¡ x ¡¡ ", Y: " ¡¡ y;
- Input Cascade: cin ¿¿ x ¿¿ y;
- Execution is from left to right.

Cascading makes C++ I/O very concise. Instead of writing three separate cout statements, we can chain them. When you write 'cin ¿¿ x ¿¿ y', the program pauses and waits for the user to enter two values separated by whitespace. The first goes into x, the second into y. This is because 'cin ¿¿ x' returns the 'cin' object itself, allowing the next '¿¿ y' to operate on it.

# Introduction to Namespaces

- Problem: Name collisions (e.g., two libraries both have a 'print' function).
- Solution: Namespaces group entities like classes, objects, and functions under a name.
- Think of it as a folder or directory for your code.
- Standard Library entities (like cin, cout, string) are in the 'std' namespace.
- Fully qualified name: std::cout





Introduction to Namespaces
- Problem: Name collisions (e.g., two libraries both have a 'print' function).
- Solution: Namespaces group entities like classes, objects, and functions under a name.
- Think of it as a folder or directory for your code.
- Standard Library entities (like cin, cout, string) are in the 'std' namespace.
- Fully qualified name: std::cout

As programs grow and you use libraries from different authors, you might encounter 'name collisions'. What if two libraries define a class called 'Vector'? Namespaces solve this by acting as a declarative region that provides a scope to the identifiers inside it. All standard C++ library components are declared within a namespace called 'std'. That is why, strictly speaking, cout is actually std::cout. The '::' is the scope resolution operator.

# The using Directive

- Writing std:: before every cout and cin can be tedious.
- The 'using namespace std;' directive brings all names from 'std' into current scope.
- Allows writing 'cout' instead of 'std::cout'.
- Syntax: using namespace std; (usually placed after #include directives).





The using Directive
- Writing std:: before every cout and cin can be tedious.
- The 'using namespace std;' directive brings all names from 'std' into current scope.
- Allows writing 'cout' instead of 'std::cout'.
- Syntax: using namespace std; (usually placed after #include directives).

To save typing, C++ provides the 'using' directive. When you place 'using namespace std;' at the top of your file, you tell the compiler to look in the std namespace if it can't find a name in the global scope. This is very common in beginner tutorials and small programs because it makes the code cleaner to read.

# Pros and Cons of using namespace std

- Pros: Reduces typing, makes simple code cleaner and easier to read for beginners.
- Cons: Defeats the purpose of namespaces (pollutes the global namespace).
- Risk of name conflicts if you define a variable or function with a standard library name (e.g., max, min).
- Best Practice for larger projects: Use 'using std::cout; using std::cin;' or just type std::.





Pros and Cons of using namespace std
- Pros: Reduces typing, makes simple code cleaner and easier to read for beginners.
- Cons: Defeats the purpose of namespaces (pollutes the global namespace).
- Risk of name conflicts if you define a variable or function with a standard library name (e.g., max, min).
- Best Practice for larger projects: Use 'using std::cout; using std::cin;' or just type std::.

While 'using namespace std;' is convenient, it is generally discouraged in professional, large-scale software engineering. It pulls thousands of names into your global scope. If you create a variable named 'count', you might accidentally conflict with std::count. A better compromise is specific using declarations, like 'using std::cout;', which only imports cout, or just embracing typing 'std::'. For this course's small examples, 'using namespace std;' is acceptable, but you should be aware of its drawbacks.

# Stream Manipulators: Overview

- Functions or objects used with the insertion (¡¡) or extraction (¿¿) operators.
- They modify the formatting state of the stream.
- They do not output data themselves; they change HOW subsequent data is output.
- Categories: No-argument manipulators (¡iostream¿) and parameterized manipulators (¡iomanip¿).





Stream Manipulators: Overview
- Functions or objects used with the insertion (¡¡) or extraction (¿¿) operators.
- They modify the formatting state of the stream.
- They do not output data themselves; they change HOW subsequent data is output.
- Categories: No-argument manipulators (¡iostream¿) and parameterized manipulators (¡iomanip¿).

Now we know how to print, but how do we make it look good? That's where stream manipulators come in. They are special objects that, when placed in the I/O stream, alter the formatting parameters of that stream. Some affect only the very next item printed, while others change the stream's state permanently until changed back.

# The endl Manipulator

- Defined in ¡iostream¿.
- Inserts a newline character ('\n') into the output stream.
- Crucially, it also 'flushes' the output buffer.
- Flushing forces the system to immediately write buffered data to the screen.
- Example: cout ¡¡ "Hello" ¡¡ endl;





The endl Manipulator
- Defined in ¡iostream¿.
- Inserts a newline character ('\n') into the output stream.
- Crucially, it also 'flushes' the output buffer.
- Flushing forces the system to immediately write buffered data to the screen.
- Example: cout ¡¡ "Hello" ¡¡ endl;

The most common manipulator is 'endl', which stands for 'end line'. It does two things: it moves the cursor to the next line (like '\n'), and it flushes the output buffer. Output operations are often buffered for performance; the system waits until a chunk of text is ready before writing to the screen. 'endl' forces the screen to update immediately. If you are doing a lot of output in a loop, printing '\n' is faster than 'endl' because it avoids constant flushing.

# Formatting Output: setw

- Defined in ¡iomanip¿ (Input/Output Manipulation library).
- setw(n): Sets the field width to 'n' characters for the NEXT output operation only.
- Useful for printing aligned columns or tables.
- Default is right-justified.
- Example: cout ¡¡ setw(10) ¡¡ "Price" ¡¡ setw(10) ¡¡ 45.5;





Formatting Output: setw
- Defined in ¡iomanip¿ (Input/Output Manipulation library).
- setw(n): Sets the field width to 'n' characters for the NEXT output operation only.
- Useful for printing aligned columns or tables.
- Default is right-justified.
- Example: cout ¡¡ setw(10) ¡¡ "Price" ¡¡ setw(10) ¡¡ 45.5;

To use parameterized manipulators like setw, we must #include ¡iomanip¿. 'setw' stands for set width. It allocates a specific number of spaces for the next item to be printed. If the item is shorter than the width, it pads it with spaces (by default, on the left, pushing text to the right). Important note: setw only applies to the immediately following data. After that, the width resets to 0.

# Formatting Output: setprecision and fixed

- setprecision(n): Sets the number of digits to display for floating-point numbers.
- By default, it specifies the total number of significant digits.
- fixed: Forces floating-point numbers to display in fixed-point notation (decimal format).
- When used with 'fixed', setprecision(n) sets the number of digits AFTER the decimal point.
- Example: cout ¡¡ fixed ¡¡ setprecision(2) ¡¡ 3.14159; // Outputs 3.14





Formatting Output: setprecision and fixed
- setprecision(n): Sets the number of digits to display for floating-point numbers.
- By default, it specifies the total number of significant digits.
- fixed: Forces floating-point numbers to display in fixed-point notation (decimal format).
- When used with 'fixed', setprecision(n) sets the number of digits AFTER the decimal point.
- Example: cout ¡¡ fixed ¡¡ setprecision(2) ¡¡ 3.14159; // Outputs 3.14

When dealing with money or scientific data, controlling decimal places is vital. 'setprecision' does this. By itself, it counts all digits. But usually, we want to specify decimal places. To do this, we send the 'fixed' manipulator first. Once a stream is set to 'fixed' and a specific precision, those settings are 'sticky'—they remain in effect for all future floating-point outputs until you change them.

# Formatting Base: hex, oct, dec

- Changes the numerical base for integer input/output.
- hex: Output integers in hexadecimal (base 16).
- oct: Output integers in octal (base 8).
- dec: Output integers in decimal (base 10) - this is the default.
- Example: int x = 255; cout ¡¡ hex ¡¡ x; // Outputs ff





Formatting Base: hex, oct, dec
- Changes the numerical base for integer input/output.
- hex: Output integers in hexadecimal (base 16).
- oct: Output integers in octal (base 8).
- dec: Output integers in decimal (base 10) - this is the default.
- Example: int x = 255; cout ¡¡ hex ¡¡ x; // Outputs ff

In computer science, we often need to look at numbers in different bases, particularly hexadecimal for memory addresses or color codes. Sending 'hex' to cout changes the internal state so all subsequent integers are printed in base 16. You can use 'dec' to revert back to standard decimal. These are also sticky manipulators.

# Other Useful Manipulators

- setfill(char): Sets the fill character used by setw (default is space). e.g., setfill('*')
- left: Left-justifies output within the field width.
- right: Right-justifies output (default).
- showpoint: Forces the decimal point and trailing zeros to display.
- boolalpha: Prints boolean values as 'true' or 'false' instead of 1 or 0.





Other Useful Manipulators
- setfill(char): Sets the fill character used by setw (default is space). e.g., setfill('*')
- left: Left-justifies output within the field width.
- right: Right-justifies output (default).
- showpoint: Forces the decimal point and trailing zeros to display.
- boolalpha: Prints boolean values as 'true' or 'false' instead of 1 or 0.

There are many other manipulators. If you use setw(10) but want to fill the empty space with dashes instead of spaces, use setfill('-'). If you want text aligned to the left side of a column instead of the right, use the 'left' manipulator. And 'boolalpha' is very handy for debugging when you want to see the words true/false instead of numbers.

# Comprehensive Code Example

- #include ¡iostream¿
- #include ¡iomanip¿
- using namespace std;
- int main() {
- double price = 12.5; int qty = 3;
- cout ¡¡ left ¡¡ setw(15) ¡¡ "Item" ¡¡ right ¡¡ setw(10) ¡¡ "Cost" ¡¡ endl;
- cout ¡¡ setfill('-') ¡¡ setw(25) ¡¡ "" ¡¡ setfill(' ') ¡¡ endl;
- cout ¡¡ left ¡¡ setw(15) ¡¡ "Widget" ¡¡ right ¡¡ setw(10)
- ¡¡ fixed ¡¡ setprecision(2) ¡¡ (price * qty) ¡¡ endl;
- return 0;
- }





Comprehensive Code Example
- #include ¡iostream¿
- #include ¡iomanip¿
- using namespace std;
- int main() {
- double price = 12.5; int qty = 3;
- cout ¡¡ left ¡¡ setw(15) ¡¡ "Item" ¡¡ right ¡¡ setw(10) ¡¡ "Cost" ¡¡ endl;
- cout ¡¡ setfill('-') ¡¡ setw(25) ¡¡ "" ¡¡ setfill(' ') ¡¡ endl;
- cout ¡¡ left ¡¡ setw(15) ¡¡ "Widget" ¡¡ right ¡¡ setw(10)
- ¡¡ fixed ¡¡ setprecision(2) ¡¡ (price * qty) ¡¡ endl;
- return 0;
- }

Let's look at this block of code. We include both iostream and iomanip. We print a header, left aligning 'Item' in a 15-character block, right aligning 'Cost' in 10. Then we use a trick: we set fill to '-', ask for a width of 25, and print an empty string to create a divider line, then reset fill to space. Finally, we print the data, forcing 2 decimal places for the cost. This creates a neatly formatted table.

# Common Pitfalls & Best Practices

- Forgetting #include ¡iomanip¿ when using parameterized manipulators.
- Forgetting that setw() is non-sticky and applies only to the next item.
- Input buffer issues: mixing cin ¿¿ with getline() can cause the program to skip inputs due to leftover newline characters.
- Best Practice: Use meaningful prompts before cin to guide the user.
- Best Practice: Validate user input (we will cover this in later chapters).





Common Pitfalls & Best Practices
- Forgetting #include ¡iomanip¿ when using parameterized manipulators.
- Forgetting that setw() is non-sticky and applies only to the next item.
- Input buffer issues: mixing cin ¿¿ with getline() can cause the program to skip inputs due to leftover newline characters.
- Best Practice: Use meaningful prompts before cin to guide the user.
- Best Practice: Validate user input (we will cover this in later chapters).

A very common student error is trying to use setw and getting a compiler error because they forgot ¡iomanip¿. Another is using setw once and expecting the whole column to align, forgetting it needs to be applied before every item in that column. Finally, when you mix standard 'cin ¿¿' with functions that read whole strings with spaces, you often get bugs where the program skips a prompt. We'll learn how to clear the input buffer to fix this later.

# Summary

- C++ uses streams for Input/Output (cin, cout).
- ¡¡ is the insertion operator; ¿¿ is the extraction operator.
- Namespaces (like std) organize code and prevent naming conflicts.
- The 'using namespace std;' directive simplifies code but should be used cautiously.
- Manipulators (¡iostream¿, ¡iomanip¿) allow powerful formatting of console output.





Summary
- C++ uses streams for Input/Output (cin, cout).
- ¡¡ is the insertion operator; ¿¿ is the extraction operator.
- Namespaces (like std) organize code and prevent naming conflicts.
- The 'using namespace std;' directive simplifies code but should be used cautiously.
- Manipulators (¡iostream¿, ¡iomanip¿) allow powerful formatting of console output.

To summarize, today we learned the core of C++ user interaction. We use cout with the insertion operator to print, and cin with the extraction operator to read data. We explored the std namespace, which holds these objects, and how to access them. Finally, we learned how to format our output beautifully using manipulators like endl, setw, fixed, and setprecision. This forms the basis of all command-line C++ programs.