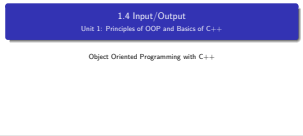
1.4 Input/Output

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

Object Oriented Programming with C++



Lecture 8: Input/Output in C++

- Unit 1: Principles of OOP and Basics of C++
- Topic 1.4: Input/Output
- cin, cout, manipulators, namespaces

Lecture 8: Input/Output in C++
- Unit 1: Principles of OOP and Basics of C++
- Topic 1.4: Input/Output
- cin, cout, manipulators, namespaces

Welcome everyone to Lecture 8. Today we dive into the fundamental ways a C++ program interacts with the user via the console. We will learn how to read data in, print data out, format that output beautifully, and handle the standard namespace.

Learning Objectives

- By the end of this lecture, you will be able to:
- 1. Understand the stream-based I/O model in C++
- 2. Use 'cout' and the insertion operator to display data
- 3. Use 'cin' and the extraction operator to read user input
- 4. Format output using manipulators like 'setw' and 'setprecision'
- 5. Explain and use namespaces, specifically the 'std' namespace

Learning Objectives
- By the end of this lecture, you will be able to:
- 1. Understand the stream-based I/O model in C++
- 2. Use 'cout' and the insertion operator to display data
- 3. Use 'cin' and the extraction operator to read user input
- 4. Format output using manipulators like 'setw' and 'setprecision'
- 5. Explain and use namespaces, specifically the 'std' namespace

Here is what we aim to achieve by the end of this session. I/O is critical because without it, our programs cannot interact with the outside world. They would just process data silently. Let's make our programs talk!

The Concept of Streams in C++

- C++ uses the concept of 'streams' to perform input and output operations.
- A stream is a sequence of bytes flowing into or out of a program.
- Input Stream: Flow of data from a source (like a keyboard) into the program.
- Output Stream: Flow of data from the program to a destination (like a monitor).
- To use standard I/O streams, we must include the `<iostream>` header file.

The Concept of Streams in C++
- C++ uses the concept of 'streams' to perform input and output operations.
- A stream is a sequence of bytes flowing into or out of a program.
- Input Stream: Flow of data from a source (like a keyboard) into the program.
- Output Stream: Flow of data from the program to a destination (like a monitor).
- To use standard I/O streams, we must include the <iostream> header file.

Think of a stream as a conveyor belt of data. Bytes go in, bytes go out. C++ abstracts the hardware details away from us. Whether we are reading from a keyboard, a file, or a network socket, the stream concept remains consistent. For basic console I/O, we always need to include the iostream header.

Standard Output: cout

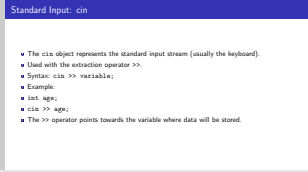
- The cout object represents the standard output stream (usually the screen).
- Used with the insertion operator <<.
- Syntax: `cout << data;`
- Example:
- `cout << "Hello, World!";`
- `int age = 20;`
- `cout << age;`
- The << operator points in the direction of data flow (towards cout).

Standard Output: cout
- The cout object represents the standard output stream (usually the screen).
- Used with the insertion operator <<.
- Syntax: cout << data;
- Example:
- cout << "Hello, World!";
- int age = 20;
- cout << age;
- The << operator points in the direction of data flow (towards cout).

The word 'cout' stands for 'character output'. The arrows or less-than signs form the 'insertion' operator. It visually points to where the data is going: the data goes into cout, and cout puts it on the screen. Notice how cout can handle strings, integers, and other basic types without us having to specify the type.

Standard Input: cin

- The `cin` object represents the standard input stream (usually the keyboard).
- Used with the extraction operator >>.
- Syntax: `cin >> variable;`
- Example:
- `int age;`
- `cin >> age;`
- The >> operator points towards the variable where data will be stored.



'cin' stands for 'character input'. Here, we use the extraction operator, which points away from cin and into our variable. It extracts data from the stream and puts it in memory. Important note: cin stops reading when it encounters whitespace (space, tab, newline).

Cascading I/O Operators

- We can chain multiple << or >> operators in a single statement.
- This is called 'cascading'.
- Output Example:
- `cout << "I am " << age << " years old.";`
- Input Example:
- `int x, y;`
- `cin >> x >> y;`
- Execution happens from left to right.

Cascading I/O Operators
- We can chain multiple << or >> operators in a single statement.
- This is called 'cascading'.
- Output Example:
- cout << "I am " << age << " years old.";
- Input Example:
- int x, y;
- cin >> x >> y;
- Execution happens from left to right.

Cascading makes our code much cleaner. Instead of writing three separate cout statements for 'I am', the age variable, and 'years old', we string them together. For input, 'cin ¿¿ x ¿¿ y' will wait for the user to type two numbers separated by a space or a new line.

Introduction to Namespaces

- A namespace is a declarative region that provides a scope to the identifiers (names of types, functions, variables) inside it.
- Purpose: To prevent name conflicts in large projects.
- Imagine two libraries both have a function named `print()`. Namespaces help the compiler distinguish which `print()` to use.
- Think of it like surnames for functions and variables.

Introduction to Namespaces
- A namespace is a declarative region that provides a scope to the identifiers (names of types, functions, variables) inside it.
- Purpose: To prevent name conflicts in large projects.
- Imagine two libraries both have a function named `print()`. Namespaces help the compiler distinguish which `print()` to use.
- Think of it like surnames for functions and variables.

As C++ programs grow, you might accidentally use the same name for two different things, especially when using third-party libraries. Namespaces solve this. If John from family Smith and John from family Doe are in the same room, you call them John Smith and John Doe to avoid confusion. Namespaces do exactly this for code.

The 'std' Namespace

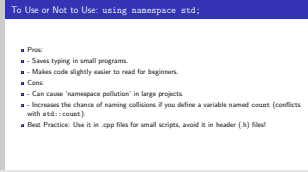
- All elements of the C++ Standard Library are declared within a namespace called `std`.
- This includes `cout`, `cin`, `endl`, string, vector, etc.
- Fully qualified name: `std::cout << "Hello";`
- The `::` is the Scope Resolution Operator.
- To avoid typing `std::` everywhere, we use the using directive: `using namespace std;`

The 'std' Namespace
- All elements of the C++ Standard Library are declared within a namespace called std.
- This includes cout, cin, endl, string, vector, etc.
- Fully qualified name: std::cout << "Hello";
- The :: is the Scope Resolution Operator.
- To avoid typing std:: everywhere, we use the using directive: using namespace std;

The C++ standard library developers put everything into the 'std' namespace to make sure they don't conflict with your custom code. You can explicitly say std::cout, which is the safest way. But for beginners and small academic programs, typing std:: every time gets tedious.

To Use or Not to Use: `using namespace std;`

- Pros:
- - Saves typing in small programs.
- - Makes code slightly easier to read for beginners.
- Cons:
- - Can cause 'namespace pollution' in large projects.
- - Increases the chance of naming collisions if you define a variable named `count` (conflicts with `std::count`).
- Best Practice: Use it in .cpp files for small scripts, avoid it in header (.h) files!



While we will use 'using namespace std;' in our class examples to keep the slides clean, you should know that in professional, large-scale software engineering, it is often discouraged, especially in header files, because it defeats the entire purpose of having namespaces by dumping all standard names into the global scope.

Formatting Output: Why?

- Default output can be messy or unaligned, especially with numbers.
- Example:
- `double pi = 3.14159265;`
- How do we print only 2 decimal places? (3.14)
- How do we align columns in a table?
- Solution: C++ I/O Manipulators.

Formatting Output: Why?
- Default output can be messy or unaligned, especially with numbers.
- Example:
- double pi = 3.14159265;
- How do we print only 2 decimal places? (3.14)
- How do we align columns in a table?
- Solution: C++ I/O Manipulators.

When we build business applications, financial software, or data tables, presentation matters. If you print a list of prices, you want them aligned by the decimal point, always showing two decimal places. C++ provides tools called manipulators to modify how the stream formats data.

Basic I/O Manipulators

- Included in `<iostream>`:
- `endl`: Inserts a newline character ('\n') and flushes the output stream.
- `cout << "Line 1" << endl << "Line 2";`
- `flush`: Flushes the output stream without adding a newline.
- `ws`: Extracts and discards whitespace characters from the input stream (`cin >> ws;`).

Basic I/O Manipulators
- Included in <iostream>:
- endl: Inserts a newline character ('\n') and flushes the output stream.
- cout << "Line 1" << endl << "Line 2";
- flush: Flushes the output stream without adding a newline.
- ws: Extracts and discards whitespace characters from the input stream (cin >> ws;).

The most common manipulator is 'endl'. It acts like pressing Enter. Note that endl also flushes the buffer, meaning it forces the program to immediately write the data to the console, which can be slightly slower than just using the newline character slash-n, but is safer if the program crashes.

Parameterized Manipulators (¡iomanip¿)

- Require `#include <iomanip>`
- `setw(n)`: Sets the field width to 'n' characters for the *next* output only.
- Example:
- `cout << setw(10) << "Hello";`
- Prints: `Hello` (5 spaces padding).
- Note: `setw` does NOT truncate if the data is larger than 'n'.

Parameterized Manipulators (¡iomanip¿)
- Require #include <iomanip>
- setw(n): Sets the field width to 'n' characters for the next output only.
- Example:
- cout << setw(10) << "Hello";
- Prints: Hello (5 spaces padding).
- Note: setw does NOT truncate if the data is larger than 'n'.

To use parameterized manipulators—those that take an argument in parentheses—we must include the iomanip header. setw stands for set width. It's incredibly useful for printing tables. Keep in mind, setw only affects the very next item outputted. After that, the width resets to 0.

Precision and Fill Manipulators

- `setprecision(n)`: Sets the number of significant digits (or decimal places if used with `fixed`).
- `cout << setprecision(3) << 3.14159; // Prints 3.14`
- `setfill(c)`: Sets the fill character used by `setw` (default is space).
- `cout << setfill('*') << setw(10) << 123;`
- Prints: `*123`

Precision and Fill Manipulators
- setprecision(n): Sets the number of significant digits (or decimal places if used with fixed).
- cout << setprecision(3) << 3.14159; // Prints 3.14
- setfill(c): Sets the fill character used by setw (default is space).
- cout << setfill('*') << setw(10) << 123;
- Prints: *123

setprecision is persistent, meaning once you set it, it applies to all future floating-point outputs until you change it again. By default, setprecision controls the total number of digits. setfill changes what character is used for padding when setw creates empty space.

Stream Formatting Flags

- Flags modify stream state persistently.
- `fixed`: Forces floating-point numbers to display in fixed-point notation (normal decimal). Changes `setprecision` to count digits *after* the decimal.
- `scientific`: Displays in scientific notation (e.g., 3.14e+00).
- `left` / `right`: Justifies output within the width set by `setw` (default is right).
- `showpoint`: Forces the decimal point to be printed even for whole numbers.

Stream Formatting Flags
- Flags modify stream state persistently.
- fixed: Forces floating-point numbers to display in fixed-point notation (normal decimal). Changes setprecision to count digits after the decimal.
- scientific: Displays in scientific notation (e.g., 3.14e+00).
- left / right: Justifies output within the width set by setw (default is right).
- showpoint: Forces the decimal point to be printed even for whole numbers.

These flags are also manipulators. When you combine 'fixed' and 'setprecision(2)', you get standard currency formatting—exactly two digits after the decimal. Also, numbers are right-justified by default, meaning they stick to the right edge of the setw field. You can change this using 'left'.

Summary

- C++ uses `<iostream>` for standard input (`cin`) and output (`cout`).
- Cascading << and >> allows multiple operations per statement.
- Namespaces, particularly `std`, help avoid naming conflicts.
- The `<iomanip>` header provides manipulators like `setw` and `setprecision`.
- Manipulators and flags give you precise control over console output formatting.

Summary
- C++ uses <iostream> for standard input (cin) and output (cout).
- Cascading << and >> allows multiple operations per statement.
- Namespaces, particularly std, help avoid naming conflicts.
- The <iomanip> header provides manipulators like setw and setprecision.
- Manipulators and flags give you precise control over console output formatting.

To wrap up: I/O in C++ is stream-based. We use cin for input, cout for output. We learned how to avoid namespace collisions with std, and most importantly, we learned how to make our text look professional and aligned using manipulators.

Next Lecture Preview

- Topic 1.5: Control Structures in C++
- Decision making: if, if-else, switch-case.
- Loops: for, while, do-while.
- Controlling execution flow with break and continue.

Next Lecture Preview
- Topic 1.5: Control Structures in C++
- Decision making: if, if-else, switch-case.
- Loops: for, while, do-while.
- Controlling execution flow with break and continue.

Next time, we will move away from linear execution and learn how to make our programs make decisions and repeat tasks using control structures.