



CC-211L Object Oriented Programming Lab

Department of Computer Science

University of the Punjab

Fall 2026

Lab 1: Environment Verification and Bitwise Operations

After completing this lab, you will be able to:

- Verify a working C++17 compiler and record its installation details.
- Compile, link and run a C++ program with warnings enabled.
- Trace the compiler's C++ header search paths.
- Inspect `<iostream>`, `<istream>` and `<ostream>` and identify how declarations are placed in `std`.
- Use bitwise operators to manipulate and inspect integer bits.

Lab duration: 1 hour 45 minutes

Part A: Environment and library exploration (30 marks)

Part B: Bitwise operations (70 marks)

Grading Scheme

Question:	1	2	3	4	5	6	7	8	Total
Points:	6	6	8	10	20	15	25	10	100
Score:									

Important instructions

- Use VS Code and its integrated terminal throughout this lab.
- Use any installed compiler that supports C++17, such as `g++`, `g++-9`, or `clang++`.
- Compile with `-std=c++17 -Wall -Wextra -pedantic` where your compiler supports these warning options.
- Part A is evidence-based. Record commands, paths and short output excerpts clearly.
- Submit one PDF containing your answers/evidence and the three Part B source files through Google Classroom.
- Screenshots must show enough of the terminal or editor to identify the command, path and result.
- Name the PDF exactly `roll_number_oop_lab1.pdf`; replace `roll_number` with your own roll number.

1 Part A: Environment and Standard-Library Exploration

Part A is an orientation and verification exercise. It is not a programming contest. The purpose is to prove that your compiler works and that you can inspect the files that support ordinary C++ input and output.

1. [6 marks] A1. Identify the compiler

Open the VS Code integrated terminal. First choose the command that names your compiler and use that command consistently. In the examples below, **CXX** means your compiler command, such as **g++**, **g++-9** or **clang++**.

On Ubuntu or another Unix-like system, you can check where it is installed with:

```
command -v CXX
CXX --version
```

On Windows PowerShell, use:

```
Get-Command CXX
CXX --version
```

Replace **CXX** with the actual command. For example, if the compiler is **g++**, run **command -v g++** on Ubuntu or **Get-Command g++** in PowerShell.

Record the following:

- a) the compiler command you used;
- b) its installation path, as reported by your terminal;
- c) the compiler version; and
- d) one sentence explaining why knowing the compiler path and version matters.

2. [6 marks] A2. Compile and run a C++17 program

Create a file named **hello.cpp**:

```
#include <iostream>

int main() {
    std::cout << "CC211 C++17 environment is ready\\n";
    return 0;
}
```

Compile and run it using your compiler command:

```
CXX -std=c++17 -Wall -Wextra -pedantic hello.cpp -o hello
./hello
```

Replace **CXX** with the command you recorded in A1. On Windows, the output file may be named **hello.exe**; run it as **.\hello.exe** in PowerShell or **./hello.exe** in a Bash-like terminal.

Submit evidence showing:

- a) the exact compile command;
- b) that compilation completed without warnings or errors; and
- c) the program's output.

3. [8 marks] A3. Find the C++ header search paths

Run the following in a Bash-like terminal, replacing **CXX** with your compiler:

```
CXX -v -E -x c++ - < /dev/null 2> include_paths.txt
```

If your VS Code terminal is Windows PowerShell, use this equivalent command:

```
'' | CXX -v -E -x c++ - 2> include_paths.txt
```

It will create a file called `include_paths.txt`. Open `include_paths.txt`. Find the section between:

```
#include <...> search starts here:  
End of search list.
```

Record:

- a) the first C++ standard-library search path
- b) the path containing your compiler's C++ standard headers, and
- c) the difference between a compiler's header search path and the compiler's executable path from part A1.

Your recorded result is the evidence that receives marks. Header paths differ between Ubuntu, Windows, compiler vendors and installation methods.

4. [10 marks] **A4. Inspect `<iostream>` and the headers it uses**

Using the first C++ standard-library path from A3, locate these files in VS Code:

```
iostream    istream    ostream
```

Use VS Code's file search or your terminal to open the files. If your terminal supports Bash commands, you may display relevant line-numbered excerpts with:

```
export CPP_INCLUDE=YOUR_RECORDED_INCLUDE_PATH  
nl -ba "$CPP_INCLUDE/iostream" | sed -n '35,68p'  
nl -ba "$CPP_INCLUDE/istream" | sed -n '35,68p'  
nl -ba "$CPP_INCLUDE/ostream" | sed -n '35,68p'
```

Submit short excerpts, or screenshots, that show:

- a) `iostream` opening the `std` namespace;
- b) the declarations of `std::cin` and `std::cout`;
- c) `istream` opening or extending the same namespace; and
- d) `ostream` opening or extending the same namespace.

Then answer:

- e) Why can several header files contribute declarations to `std`?

Important: standard-library implementations differ between compiler versions and operating systems. The required evidence is from your own C++17 compiler installation.

2 Part B: Bitwise Operations

Bitwise operators work directly on the individual bits of integer operands. They are extremely useful for tasks such as flag management, low-level data manipulation, optimization, and encoding information compactly. The table below summarizes the commonly used bitwise operators.

Operator	Syntax	Purpose	Example
AND	<code>a & b</code>	Sets each result bit to 1 only if the corresponding bits of both operands are 1. Often used to check or clear specific bits.	<code>5 & 3 → 0101 & 0011 = 0001</code> (1)
OR	<code>a b</code>	Sets each result bit to 1 if at least one of the corresponding bits is 1. Commonly used to set specific bits.	<code>5 3 → 0101 0011 = 0111</code> (7)
XOR	<code>a ^ b</code>	Sets each result bit to 1 only if the corresponding bits differ. Useful for toggling bits and swapping values without a temporary variable.	<code>5 ^ 3 → 0101 ^ 0011 = 0110</code> (6)
NOT	<code>~a</code>	Inverts every bit of the operand (1's complement). Frequently used to build masks.	<code>~5 on 8 bits: 0000 0101 → 1111 1010</code>
Left Shift	<code>a << n</code>	Shifts all bits of <code>a</code> to the left by <code>n</code> positions, filling with 0 on the right. Equivalent to multiplying by 2^n .	<code>3 << 2 → 0011 << 2 = 1100</code> (12)
Right Shift	<code>a >> n</code>	Shifts all bits of <code>a</code> to the right by <code>n</code> positions. Equivalent to dividing by 2^n (for unsigned/positive values).	<code>12 >> 2 → 1100 >> 2 = 0011</code> (3)

In this part, use integer variables and ordinary console input/output. Do not use classes, arrays, templates, STL containers, exceptions or other later-course mechanisms.

5. [20 marks] **B1. Number swap**

Create `number_swap.cpp`. Ask the user for two integer values and swap them using an appropriate bitwise operator.

Requirements:

- Do not use a third or temporary variable.
- Do not use arithmetic operators to perform the swap.
- Display the values before and after the swap.

Test with at least:

```
first = 15, second = 28
```

Expected result format:

```
Before swap: first = 15, second = 28
After swap:  first = 28, second = 15
```

Compile with:

```
CXX -std=c++17 -Wall -Wextra -pedantic number_swap.cpp -o number_swap
```

Replace CXX with the compiler command recorded in A1.

6. [15 marks] **B2. Even or odd**

Create `parity.cpp`. Read an integer and determine whether it is even or odd without using the modulus operator %.

Requirements:

- Use only bitwise operations
- Do not use % for the decision.
- Test both an even and an odd input.

Suggested tests:

```
42 -> 42 is even.
37 -> 37 is odd.
```

Written check: Explain why your solution distinguishes even and odd non-negative integers.

7. [25 marks] **B3. Display and count the bits of one byte**

Create `byte_bits.cpp`. Read an integer from 0 through 255 and display exactly its 8-bit representation, from the most-significant bit to the least-significant bit. Then count the number of zero bits and one bits.

Requirements:

- Reject or clearly report an input outside the range 0–255.
- Use a mask and bitwise operators to inspect the eight positions.
- Do not use `std::bitset`, string conversion, or a library routine that directly prints binary.
- Print exactly eight binary digits, including leading zeroes.

For input 23, the expected result is:

```
Binary: 00010111
Number of 1 bits: 4
Number of 0 bits: 4
```

Use at least one additional test, such as 0, 1, 128, or 255.

Written check: State the mask used to extract the first bit and explain how the mask moves to inspect the remaining seven positions.

8. [10 marks] **B4. Code quality and evidence**

For each Part B program:

- include a complete source file;
- compile with C++17, `-Wall`, `-Wextra` and `-pedantic`;
- show at least one successful run; and
- use clear names and readable output.

Submission checklist

Submit one PDF and the three source files ONLY. **Do not submit any executable files.** The PDF must contain:

- your full name, roll number, operating system, compiler command
- Part A answers and compiler/header evidence;
- the tested commands and outputs for A2–A4;
- the written checks for B1–B3; and
- a short note identifying any setup problem you encountered and how you resolved it.

Upload the PDF and the three source files to the Lab 1 assignment in Google Classroom. Name the PDF exactly `roll_number_oop_lab1.pdf`, using your own roll number. For example, roll number bcsf25m123 must use `bcsf25m123_oop_lab1.pdf`.

Grading scheme

Part A: environment and header exploration	30
B1: Number swap	20
B2: Parity check	15
B3: Eight-bit display and counting	25
B4: Code quality and evidence	10
Total	100

Suggested timing

A1–A3: compiler setup and paths	20 minutes
A4–A5: header inspection	20 minutes
B1: Number swap	20 minutes
B2: Parity	15 minutes
B3: Byte display and counting	25 minutes
Evidence packaging and submission	5 minutes
Total	105 minutes

Notes

- Header paths and line numbers are installation-dependent. Marks are based on evidence from the student's own compiler.
- Part A is about understanding and proving the environment, not memorizing one machine's directory layout.
- The number swap is included as a bitwise exercise; in production code, a clear temporary variable is often preferable.