Chapter 6: Comprehensive I/O and Streams

The C++ I/O library, known as `iostreams`, is a powerful, type-safe, and extensible framework for handling input and output. Its object-oriented design allows the same code to work with the console, files, and in-memory strings.

6.1 The Stream Hierarchy and Design

Why is it a hierarchy? This design is a prime example of object-oriented programming, specifically the principle of separating concerns. Each layer in the hierarchy has a distinct job, which makes the whole system flexible and extensible.

- `ios_base`: The root of the hierarchy. It knows nothing about data types or where the data is going. Its only job is to manage **formatting state** (e.g., number base, float precision) and **error state** (the stream status bits).
- `basic_ios`: Inherits from `ios_base`. It is a class template that manages the connection to the underlying **stream buffer**, which is the object that actually reads/writes to the physical device.
- `basic_istream` & `basic_ostream`: Inherit from `basic_ios`. These classes provide the high-level, type-safe formatting capabilities via the overloaded `>>` (extraction) and `<<` (insertion) operators.
- `basic_iostream`: Inherits from both `istream` and `ostream` for read/write streams.

This looks like: `ios_base` <— `basic_ios<T>` <— `basic_istream<T>` & `basic_ostream<T>` <— `basic_iostream<T>`

The familiar `std::cout` is just a type alias for `basic_ostream<char>`.

Design Insight: Buffering and `std::endl`

Why buffer? Physical I/O operations (writing to a console, a file on disk, or a network socket) are thousands of times slower than writing to memory. For performance, it would be disastrous to perform a physical write for every single character. **Buffering** solves this by collecting output in a fast in-memory buffer. The slow physical write is only performed when the buffer is full, or when explicitly told to do so.

What is `std::endl` really doing? * `stream << '\n';` - This simply adds a newline character to the stream's buffer. It's fast and efficient. * `stream << std::endl;` - This does two things: it adds a newline character to the buffer, and then it **forces a flush** of the buffer, triggering the slow physical write operation immediately.

In a tight loop printing thousands of lines, using `std::endl` can make the program dramatically slower than using `' '`. Unless you absolutely need to guarantee the output is visible on the screen *right now*, prefer `' '`.

6.2 Stream State, Manipulators, and Error Handling

Why have stream states? I/O is fragile. A user can enter invalid data, a file might not exist, a disk can be full. The stream state flags are the mechanism for detecting and handling these errors gracefully instead of crashing.

What are the states? Every stream has a set of status bits: * `goodbit`: No errors. All is well. * `eofbit`: The end of the input has been reached. This is not an error; it's expected when reading a file. * `failbit`: A recoverable formatting error. Example: trying to read `"abc"` into an `int`. The stream is now in a failed state, and further operations will do nothing until the state is cleared (with `cin.clear()`). * `badbit`: A serious, non-recoverable error with the underlying device (e.g., disk read error).

How to handle errors: The most robust way to read input is to use the stream itself as the condition.

```cpp
#include <iostream>

int value;
// The `while (std::cin >> value)` trick works because the `>>` operator returns the stream,
// and the stream object itself can be evaluated as a boolean. It's `true` if the stream is good.
while (std::cin >> value) {
    std::cout << "You entered: " << value << std::endl;
}

// After the loop, check WHY it ended.
if (std::cin.eof()) {
    std::cout << "End of input reached (e.g., Ctrl+D)." << std::endl;
} else if (std::cin.fail()) {
    std::cout << "Invalid input. Not an integer." << std::endl;
    // You would typically clear the error state and ignore the bad input here.
}
```

Manipulators

Why use them? They provide a clean, chainable syntax for modifying a stream's formatting state, which is more elegant than calling member functions like `cout.setf()`.

```
#include <iostream>
#include <iomanip> // Required for manipulators with arguments

std::cout << "| " << std::setw(10) << std::left << "Product" << " |"
          << "| " << std::setw(8) << std::right << "Price" << " |\n";

std::cout << "| " << std::setw(10) << std::left << "Apples" << " |"
          << "| " << std::setw(8) << std::right << std::fixed << std::setprecision(2) << 1.99 << " |\n";
```

6.3 File I/O and String Streams

Why are these so useful? The power of the iostream library is that the same `<<` and `>>` operators you use for the console work identically for files and in-memory strings. This provides a unified, consistent interface for I/O.

File I/O (`<fstream>`)

`std::ifstream` (input), `std::ofstream` (output), and `std::fstream` (input/output) are RAII wrappers around file handles. When an fstream object is created, it opens the file. When it is destroyed (goes out of scope), it automatically closes the file.

```
#include <fstream>
#include <string>

void write_and_read_file() {
    // Writing to a file
    std::ofstream outfile("my_data.txt");
    if (!outfile.is_open()) { // Always check if the file was opened successfully!
        std::cerr << "Error: Could not open file for writing.\n";
        return;
    }
    outfile << "This is line 1.\n";
    outfile << "This is line 2.\n";
} // outfile goes out of scope here and its destructor automatically closes the file.
```

String Streams (`<sstream>`)

`std::stringstream` is an incredibly useful tool that applies the stream interface to an in-memory `std::string`. It's perfect for parsing complex strings or building up a string from various data types.

How to parse with it:

```
#include <sstream>
#include <string>

// Example: Parsing a CSV (Comma Separated Value) line
std::string csv_line = "John Doe,42,185.5";
std::stringstream ss(csv_line);

std::string name;
int age;
double height;
char comma;

// std::getline can read until a delimiter character
std::getline(ss, name, ',');
ss >> age >> comma >> height; // Read age, then consume the next comma

std::cout << "Parsed: Name='" << name << "', Age=" << age << ", Height=" << height << std::endl;
```

Projects for Chapter 6

Project 1: The Formatted Receipt Generator

- **Problem Statement:** Write a program that defines several items with names and prices (e.g., in a `struct` or `std::vector<std::pair<std::string, double>>`). Your program should then print a neatly formatted receipt to the console. Use I/O manipulators from `<iomanip>` to ensure that item names are left-aligned in a column of a fixed width, and prices are right-aligned in another column, with exactly two digits of precision.
- **Core Concepts to Apply:** `std::cout`, manipulators (`std::setw`, `std::setfill`, `std::left`, `std::right`, `std::fixed`, `std::setprecision`).

Project 2: The Log File Writer

- **Problem Statement:** Create a simple logging system. Write a function `void log_message(const std::string& message)` that appends the given message, preceded by a timestamp, to a file named `app.log`. Implement the timestamp by getting the current time (you can use the `<chrono>` and `<ctime>` headers for this). Ensure that each call to `log_message` opens the file in append mode, writes the message, and closes it.
- **Core Concepts to Apply:** `std::ofstream`, opening files in append mode (`std::ios::app`), RAII for file handles.
- **Hint:** `std::ofstream logfile("app.log", std::ios::app);`

Project 3: The Configuration File Parser

- **Problem Statement:** You have a simple configuration file (`config.txt`) with key-value pairs, like: `# This is a comment user = jsmith level = 12 fullscreen = true` Write a program that reads this file line by line. It should ignore empty lines and lines starting with `#`. For valid lines, it should use a `std::stringstream` to parse the key and the value separated by the `=` sign and print them out, e.g., `"Key: 'user', Value: 'jsmith'"`.
- **Core Concepts to Apply:** `std::ifstream`, `std::getline` to read the file, `std::stringstream` to parse each line.

“‘