Input and Output

The Art of Computer Programming

Pierre Senellart

PSL UNIVERSITÉ PARIS

8 December 2025

Outline

Why Input/Output?

- Most useful programs do not just compute; they interact:
  - Read input from the user, from files, from the network
  - Write results, logs, and error messages
- Input/output (I/O) is the boundary between the program and the outside world
- Large software programs may have complex input/output (graphical interfaces, audio, capturing various input devices, etc.) – here, we focus on simple text-based input and output
- Good understanding of I/O requires thinking beyond syntax of one specific programming language:
  - How the operating system represents files and devices
  - Performance issues (buffering)
  - Safety (errors, invalid input, resource management)

Returning a Value vs Producing Output

- A function can return a value to its caller:
  - This value is used inside the program
  - It participates in further computations
  - It is part of the program's logic, not its user interface
- A function can also produce output:
  - Writing to `stdout`, `stderr`, or a file
  - This communicates with the outside world
  - Output does not influence the program's internal computation
- Confusing the two leads to poor design:
  - Functions that both compute *and* print are harder to reuse and test
  - Returning a value is part of the program's data flow
  - Printing is a side effect, not a computation
- Good practice:
  - Functions should return values; callers decide what to do with them
  - Output should happen at well-defined places (main logic, UI layer)

Standard Streams

- Most systems and languages provide three standard streams:

Standard input (`stdin`)	Where the program usually reads from (keyboard, or data piped from another program)
Standard output (`stdout`)	Where normal program results are written
Standard error (`stderr`)	Where error messages and diagnostics go

- Advantages:
  - Can be redirected independently (e.g., to files)
  - Programs can be composed with pipes
- In our three languages:

C:	`stdin`, `stdout`, `stderr` (`<stdio.h>`)
C++:	`std::cin`, `std::cout`, `std::cerr` (`<iostream>`)
Python:	`sys.stdin`, `sys.stdout`, `sys.stderr` (`import sys`)

The Pipeline Philosophy

- On Linux and macOS, programs are often designed to:
  - Read from standard input (`stdin`)
  - Write results to standard output (`stdout`)
  - Report errors to standard error (`stderr`)
- This allows programs to be combined:

Bash

```
cat data.txt | sort | uniq -c | sort -nr
```

  - Each program does one thing well
  - Output of one becomes input of the next
- Advantages:
  - Powerful data processing using small, reusable tools
  - Programs remain independent of user interface choices
  - Encourages separation of computation and presentation
- Also works under Windows, though not as common
- A well-designed program:
  - Does not assume input comes from a keyboard
  - Does not assume output goes to a screen

Files as Resources

- A file is a sequence of bytes stored by the operating system
- To use a file, a program must:
  1. Open it: ask the OS for permission/access
  2. Read or write data
  3. Close it: release the resource
- The operating system only allows a limited number of files to be open at the same time
- Forgetting to close files can lead to:
  - Resource exhaustion (program cannot open more files)
  - Data not being written to disk properly (because of buffering, see further)

Outline

Files, Devices, and Paths

- In Unix-like systems (e.g., Linux, macOS), "everything is a file":
  - Regular files on disk
  - Devices (terminals, disks, pseudo-random number generator, etc.)
  - Named pipes (files that one can write to from one process and read from another) and sockets (network connections)
- In Windows, files on disk are distinct from many devices; some devices exist (e.g., `CON`, `NUL`) but they do not participate in a unified file hierarchy
- A path (such as `/home/toto/data.txt` in Unix-like systems or `C:/Users/Toto/Desktop/`, often written `C:\Users\Toto\Desktop\`, under Windows) identifies a file in the file system
- The program itself does not access the disk directly: it asks the operating system to read or write portions of the file

File Descriptors and Handles

- At the OS level, an open file is identified by a small integer called a file descriptor (FD); By convention: 0 = stdin, 1 = stdout, 2 = stderr
- Higher-level libraries wrap file descriptors:
  - C: `FILE *` (buffers, formatted I/O)
  - C++: `std::ifstream`, `std::ofstream`, `std::fstream`
  - Python: file objects returned by open

C

```c
#include <fcntl.h>
#include <unistd.h>

int main(void) {
  int fd = open("test.txt", O_RDONLY); // low-level FD
  if (fd == -1) { return 1; }
  char buffer[16];
  int n = read(fd, buffer, sizeof(buffer));
  if(n>0) write(1, buffer, n);
  close(fd); // important!
}
```

Buffering

- If every single byte written by a program went straight to disk, I/O would be very slow
- Solution: buffering
  - Data is stored in memory and written in larger chunks
  - Similarly, input can be read from disk in larger chunks and then consumed gradually by the program
- Typical modes:

Unbuffered	Every operation goes directly to the device (rare)
Line-buffered	Flush when a newline is written (common for terminals)
Block-buffered	Flush when buffer is full or explicitly flushed

- Consequences:
  - Data may not appear immediately on screen or in files
  - Flushing becomes important, especially before a program crashes

Flushing Buffers

- In all three languages, one can force buffered output to be written

C

```c
#include <stdio.h>

printf("Processing...");
// make sure the message appears now
fflush(stdout);
```

C++

```cpp
#include <iostream>

std::cout << "Processing..." << std::flush; // or std::endl (also adds
↪  '\n')
```

Python

```python
print("Processing...", flush=True)
# or:
import sys
print("Processing...", file=sys.stdout)
sys.stdout.flush()
```

A Key Platform Difference: Text vs Binary Mode

- On Linux and macOS:
  - Regular files are just sequences of bytes
  - Writing \n writes the newline byte, $\underline{0a}_{16}$
  - No difference between text mode and binary mode
- On Windows:
  - By default, opening a file in text mode modifies the data
  - Writing \n becomes \r\n (two bytes, $\underline{0d0a}_{16}$)
  - Reading \r\n becomes \n
- Recommendation:
  - When handling binary data (images, protocols, etc.), always specify files are binary, e.g., in C:

C

```c
FILE *f = fopen("data.bin", "rb"); // or "wb"
```

- You also need to careful about:
  - Encodings in text mode
  - Cross-platform consistency when sharing files

Outline

Two Ways to Read Input

- Conceptually, there are two main styles:

  Line-oriented Read a whole line of text, then parse it

  Token-oriented Read values of specific types directly (formatted input)

- Line-oriented:
  - Keeps input as a character string
  - Parsing is explicit and under your control
  - Easier to handle malformed input robustly
- Token-oriented:
  - Convenient: directly read int, double, etc.
  - But parsing rules can be subtle
  - Error handling can be more complex

Example: Read an Integer (Line-Oriented, C)

C

```c
#include <stdio.h>
#include <stdlib.h>

int main(void) {
  char line[100];
  if (!fgets(line, sizeof(line), stdin)) {
    // input error or EOF
    return 1;
  }
  char *end;
  long x = strtol(line, &end, 10);
  if (end == line) {
    printf("Not a number\n");
  } else {
    printf("You entered %ld\n", x);
  }
}
```

Example: Read an Integer (Line-Oriented, C++)

C++

```cpp
#include <iostream>
#include <string>

int main() {
  std::string line;
  if (!std::getline(std::cin, line)) return 1;
  try {
    int x = std::stoi(line);
    std::cout << "You entered " << x << "\n";
  } catch (const std::exception &) {
    std::cout << "Not a number\n";
  }
}
```

Example: Read an Integer (Line-Oriented, Python)

Python

```python
line = input("Enter an integer: ")
try:
    x = int(line)
    print("You entered", x)
except ValueError:
    print("Not a number")
```

- In Python, `input` always returns a string
- We explicitly convert to `int` and handle errors with exceptions

Example: Read an Integer (Token-Oriented, C)

C

```c
#include <stdio.h>

int main(void) {
  int x;
  if (scanf("%d", &x) != 1) { // token-oriented
    printf("Failed to read integer\n");
  } else {
    printf("You entered %d\n", x);
  }
}
```

Example: Read an Integer (Token-Oriented, C++)

C++

```cpp
#include <iostream>

int main() {
  int x;
  if (std::cin >> x) { // token-oriented
    std::cout << "You entered " << x << "\n";
  } else {
    std::cout << "Failed to read integer\n";
  }
}
```

- Convenient, but we must always check whether input succeeded

Outline

C Standard I/O Overview

- Header: `<stdio.h>`
- Central abstraction: `FILE *` (a stream, often buffered)
- Typical operations:
  - Open: `fopen`, close: `fclose`
  - Output: `fprintf`, `fputs`, `fputc`
  - Input: `fscanf`, `fgets`, `fgetc`
- Standard streams:
  - `stdin`, `stdout`, `stderr`
- Errors:
  - Functions return special values on error
  - Often set the global variable `errno`

C: Opening and Closing Files

C

```c
#include <stdio.h>

int main(void) {
  FILE *f = fopen("output.txt", "w");
  if (f == NULL) {
    perror("fopen"); // prints human-readable error
    return 1;
  }

  fprintf(f, "Hello, world!\n");
  if (fclose(f) != 0) {
    perror("fclose");
    return 1;
  }
  return 0;
}
```

- Always check that `fopen` did not return `NULL`
- Always close the file when done

C: Reading Lines Safely

C

```c
#include <stdio.h>

int main(void) {
  char line[256];

  while (fgets(line, sizeof(line), stdin)) {
    // line contains at most 255 chars + '\0'
    printf("Read line: %s", line);
  }

  if (ferror(stdin)) {
    perror("stdin error");
  }
  return 0;
}
```

- `fgets` avoids buffer overflows: it never writes more than the given size
- It returns `NULL` on error or end-of-file

C: Formatted Input and Its Pitfalls

C

```c
#include <stdio.h>

int main(void) {
  char name[10];
  // scanf("%s", name); // DANGEROUS: may overflow if too long
  // Safer: limit the number of characters
  if (scanf("%9s", name) != 1) {
    printf("Failed to read name\n");
    return 1;
  }
  printf("Hello, %s!\n", name);
}
```

- `scanf` is convenient but easy to misuse
- You must:
  - Provide the correct format string
  - Limit the number of characters for strings
  - Always check the return value

Format Strings

- In C, formatted I/O uses format strings to describe how data should be interpreted or printed:

C

```c
scanf("%d %lf", &i, &x);
printf("i = %d, x = %.2f\n", i, x);
```

- Power and risks:
  - Pros: very flexible, convert text to typed data
  - Cons: a mismatch between format and arguments leads to undefined behavior
- Security implications:
  - Incorrect or unvalidated format strings can cause buffer overflows or memory corruption
  - Always specify maximum field widths for strings (`"%9s"`)

Common Format Specifiers in C

Specifier	Meaning
`%d`, `%i`	Signed decimal integer (`int`)
`%u`	Unsigned decimal integer (`unsigned`)
`%x`, `%X`	Unsigned integer in hexadecimal
`%o`	Unsigned integer in octal
`%f`, `%F`	Decimal floating point (`double`)
`%e`, `%E`	Scientific notation
`%g`, `%G`	Use %f or %e depending on value
`%c`	Single byte (`char`)
`%s`	Null-terminated byte sequence (`char *`)
`%p`	Pointer value (address)
`%ld`, `%lld`	`long`, `long long` integers
`%zu`	`size_t` (for sizes and indexing)

C: Low-Level File Descriptors

C

```c
#include <fcntl.h>
#include <unistd.h>
#include <stdio.h>

int main(void) {
  int fd = open("data.bin", O_WRONLY | O_CREAT | O_TRUNC, 0644);
  if (fd == -1) { perror("open"); return 1; }

  const char msg[] = "Hi\n";
  ssize_t n = write(fd, msg, sizeof(msg));
  if (n == -1) { perror("write"); close(fd); return 1; }
  close(fd);
}
```

- Direct interface to the OS
- No formatted I/O, minimal buffering
- Used for performance or system programming

Outline

C++ Streams

- Header: `<iostream>`
- Main objects:
  - `std::cin` (input)
  - `std::cout` (output)
  - `std::cerr` (unbuffered error output)
- Use the insertion (<<) and extraction (>>) operators:
  - Type-safe
  - Automatically formatted
- Streams maintain an internal state to signal errors

C++: Simple Console I/O

C++

```cpp
#include <iostream>
#include <string>

int main() {
  std::cout << "Enter your name: ";
  std::string name;
  if (!(std::cin >> name)) {
    std::cerr << "Failed to read name\n";
    return 1;
  }
  std::cout << "Hello, " << name << "!\n";
}
```

- `std::cin >> name` reads up to whitespace
- If reading fails, the stream enters a failed state

C++: Checking Stream State (1/2)

C++

```cpp
#include <iostream>
using namespace std;
int main() {
  int x;
  while (true) {
    cout << "Enter an integer (or Ctrl+D): ";
    if (!(cin >> x)) {
      if (cin.eof()) {
        cout << "\nEnd of input\n"; break;
      } else {
        cerr << "Invalid input, clearing state\n";
        cin.clear();                // clear error flags
        cin.ignore(1000, '\n');     // discard remainder of line
      }
    } else {
      cout << "You entered " << x << "\n";
    }
  }
}
```

C++: Checking Stream State (2/2)

- `std::cin.fail()` and related methods tell us if something went wrong
- We must both clear the error state and discard problematic input

C++: File Streams and RAII

C++

```cpp
#include <fstream>
#include <iostream>

int main() {
  std::ofstream out("output.txt");
  if (!out) {
    std::cerr << "Could not open file\n";
    return 1;
  }

  out << "Hello, file!\n";
  // out is automatically closed when it goes out of scope
}
```

- `std::ifstream`, `std::ofstream`, `std::fstream` wrap a file descriptor
- Thanks to RAII, the file is closed automatically in the destructor
- This provides safety even if exceptions are thrown

C++: Buffering and Flushing

C++

```cpp
#include <iostream>

int main() {
  std::cout << "Processing..." << std::flush;
  // do some long computation
  // output appears immediately even if no newline
}
```

- `std::endl` is equivalent to `"\n" << std::flush`
- Flushing too often can hurt performance
- `std::cout` is usually tied to `std::cin` (flushed before reads); can be changed with `tie(nullptr)`

Outline

Python and I/O

- Python proposes a high-level view of I/O:
  - Built-in `print` and `input`
  - File objects with methods like `read`, `write`
- Under the hood, it still uses:
  - OS file descriptors
  - Buffers
  - Encodings
- Errors are reported through exceptions

Python: Standard Input and Output

Python

```python
name = input("Enter your name: ")  # reads a line from stdin
print("Hello,", name)              # writes to stdout
```

Python

```python
import sys

sys.stderr.write("This is an error message\n")
```

- `input` always returns a string (without the trailing newline)
- `print` adds a newline by default and can write to another stream

Parsing Input in Python

- Unlike C/C++, Python never parses automatically
- Converting explicitly ensures safety:

Python

```
line = input()
try:
    x = int(line)          # parse integer
    y = float(line)        # parse floating-point
except ValueError:
    print("Invalid number")
```

- Parsing multiple values:

Python

```
line = input()
a, b = line.split()   # split by whitespace
a = int(a)
b = int(b)
```

- Advantage: clear separation between reading text and validating its format

Formatting Output in Python

- Several ways to build formatted strings:

Concatenation: simple, but may be inefficient and hard to read

Python

```
name = "Alice"
score = 42
print("Name: " + name + ", Score: " + str(score))
```

C-style format strings: older and error-prone

Python

```
print("Name: %s, Score: %d" % (name, score))
```

`str.format`: more explicit and flexible

Python

```
print("Name: {}, Score: {}".format(name, score))
```

f-strings: (Python 3.6+) recommended modern style

Python

```
print(f"Name: {name}, Score: {score}")
```

- f-strings are preferred: fast, readable, and allow inline expressions

Python: Files and Context Managers

Python

```python
# Text file
with open("data.txt", "w", encoding="utf-8") as f:
    f.write("Hello, file!\n")

# Binary file
with open("image.bin", "rb") as f:
    chunk = f.read(1024)
```

- open returns a file object
- The `with` statement ensures the file is closed automatically even if an exception occurs
- In text mode, Python decodes bytes using an encoding (default depends on the system)

Text vs Bytes: Encodings Matter

- Files store bytes, but we want to read and write text (characters)
- To convert between text and bytes, Python uses an encoding (mapping between characters and byte values)
  - Common example: UTF-8 (Unicode)
- If the wrong encoding is used:
  - Reading fails (`UnicodeDecodeError`)
  - Text is garbled (mojibake)
  - Data may be silently corrupted

Python

```
# Always specify encoding explicitly in text mode:
with open("data.txt", "r", encoding="utf-8") as f:
    text = f.read()
```

- Binary mode (`"rb"`, `"wb"`):
  - Reads/writes raw bytes without decoding

Python: Iterating Over Files

Python

```python
with open("data.txt", encoding="utf-8") as f:
    for line in f:        # reads line by line
        line = line.rstrip("\n")
        print("Read:", line)
```

- File objects are iterators over lines
- Efficient: uses internal buffering, does not read whole file at once

Python: Buffering and Flushing

Python

```python
# Line-buffered text output, typically for terminals
print("Processing...", end="", flush=True)

# Controlling buffering for files
f = open("log.txt", "w", buffering=1)  # line-buffered
f.write("Start\n")
f.flush()  # force write
f.close()
```

- The `buffering` parameter of `open` controls buffering
- The `flush` argument of `print` forces immediate flush

Outline

Failures Are Normal

- When dealing with I/O, many things can go wrong:
  - File not found, permission denied
  - Disk full
  - Network connection lost
  - User enters invalid data
- Good code must:
  - Detect I/O errors
  - Report them clearly to the user or caller
  - Clean up resources (files, memory) in all cases
- Never assume I/O "just works"

Error Handling Patterns

C

```c
FILE *f = fopen("data.txt", "r");
if (f == NULL) {
  perror("fopen"); // report and handle
  return 1;
}
```

C++

```cpp
std::ifstream f("data.txt");
if (!f) {
  std::cerr << "Could not open file\n";
}
```

Python

```python
try:
    with open("data.txt") as f:
        process(f)
except OSError as e:
    print("I/O error:", e)
```

Safe Input Handling

- Prefer line-based input + explicit parsing when:
  - Input format is complex or evolving
  - You need to give good error messages
- Validate all external data:
  - Range checks for numbers
  - Length checks for strings
  - Allowed characters, formats (e.g., dates)
- Avoid mixing multiple input mechanisms that share the same stream in surprising ways (e.g., `scanf` and `fgets` on `stdin`)

Resource Safety

- Always release resources when done:
  - C: `fclose`, `close`
  - C++: rely on RAII (objects that close in their destructor)
  - Python: `with open(...) as f: ...`
- Design your code so that:
  - Any early return or error still closes files
  - You do not keep more open files than necessary
- Treat files and other I/O resources like heap memory: they must be properly owned and eventually released

Outline

Summary

- I/O connects programs to the outside world via streams: standard input/output/error and files
- The operating system represents open files with file descriptors; languages build higher-level abstractions on top
- I/O is usually buffered for performance: flushing matters for correctness and user experience
- Two main reading styles:
  - Line-oriented: flexible, explicit parsing
  - Token-oriented: convenient but needs careful error handling
- Safety requires:
  - Checking for errors (return codes, stream state, exceptions)
  - Managing resources (closing files, using RAII/context managers)
  - Validating external input

Licensing