CS 112

Introduction to Data Structures

WEEK 02

File I/O, Exceptions, 2D Arrays

Eric Araújo

Calvin University · Fall 2026

This Week

1. Streams: console and files
2. Opening, reading, closing
3. End of file: how to detect it **KEY**
4. Writing to a file
5. Exceptions: `throw` , `try` , `catch`
6. Stack unwinding
7. Two-dimensional arrays

Everything Is a Stream

Console

```cpp
#include <iostream>

int    n;
cin  >> n;      // in
cout << n;      // out
```

File

```cpp
#include <fstream>

ifstream fin("data.txt");
fin  >> n;      // in
ofstream fout("out.txt");
fout << n;      // out
```

> Same operators, different source. Learn one, you've learned both, a file stream just reads bytes from disk instead of the keyboard.

Opening a File

```
#include <fstream>
#include <cassert>

ifstream fin("numbers.txt");     // open on construction
assert(fin.is_open());           // ALWAYS check

ifstream fin2;                   // or open later
fin2.open("numbers.txt");

fin.close();                     // done with it
```

A file that isn't there does **not** crash your program, the stream just quietly fails. Check `is_open()` or you'll debug a "wrong answer" that's really a missing file.

TALK TO YOUR NEIGHBOR · TTYN

Which class reads *from* a file?

A. `fstream`

B. `ifstream`

C. `instream`

D. `fin`

TALK TO YOUR NEIGHBOR · TTYN

One way to open for reading is `ifstream fin("file");`: what is another?

A. `ifstream fin.open("file");`

B. `ifstream open("file") as fin;`

C. `ifstream fin;` then `fin.open("file");`

D. `ifstream fin().open("file");`

Reading: Words vs. Lines

>> reads one word

```
string word;
fin >> word;
// stops at whitespace
```

getline reads the rest

```
string line;
getline(fin, line);
// stops at newline
```

Both *report failure the same way*: the stream goes into a fail state, and testing the stream in a condition gives `false`.

Detecting End of File

```
string line;
while (getline(fin, line)) {      // read, THEN test
    cout << line << endl;
}
```

> The read has to happen **before** the test. A stream only knows it hit the end *after* a read fails; there is no "am I at the end?" you can trust beforehand.

TALK TO YOUR NEIGHBOR · TTYN

What is wrong with this loop?

```
string s;
while (true) {
    getline(fin, s);
    cout << s << endl;
    if (!fin) { break; }
}
```

A. No lines are ever processed

B. It loops forever

C. The first line is skipped

D. The last line is skipped

E. The last line is processed twice

Writing to a File

```
ofstream fout("results.txt");
assert(fout.is_open());

for (int i = 0; i < n; ++i) {
    fout << names[i] << "," << scores[i] << "\n";
}
fout.close();       // flushes buffered output to disk
```

- Opening for writing **erases** the file: use `ios::app` to append instead
- `endl` also flushes; `"\n"` is cheaper in a loop
- Forgetting `close()` can leave the last writes in the buffer

`\n`, `endl`, `flush`

Output is **buffered**: characters wait in memory until something pushes them out.

You write	Newline?	Flushes?	Use it when
`<< "\n"`	yes	**no**	almost always
`<< endl`	yes	**yes**	you need it on disk *now*
`<< flush`	**no**	yes	mid-line prompts, progress dots

`endl` is exactly `"\n"` followed by `flush`. That is the whole difference.

Proof, in Bytes

Print one word, then sleep 3 seconds. Check the file *while the program is still running*:

Program prints	File after 1 second	After the program exits
`"hello" << "\n"`	**0 bytes**, nothing yet	6 bytes
`"hello" << endl`	**6 bytes**, `hello\n`	6 bytes
`"hello" << flush`	**5 bytes**, `hello` (no newline)	5 bytes

On a **terminal** you would see no difference: consoles are line-buffered, so `"\n"` flushes anyway. Redirect to a **file** and the buffer holds everything until it fills, you flush, or the program ends. That is why a crashed program often loses its last output.

When a Function Can't Do Its Job

A function that hits bad input has three bad options and one good one.

- ~~Print an error~~: libraries shouldn't decide how to talk to the user
- ~~Return a magic value~~: `-1` is a real answer sometimes
- ~~Carry on regardless~~: the caller never learns anything went wrong
- **Throw an exception**: the caller decides what to do

> An exception is a function's way of saying: *I can't finish, and this is why.*

`throw` and `catch`

```
double squareRoot(double x) {
    if (x < 0) {
        throw invalid_argument("negative input");
    }
    return sqrt(x);
}

try {
    cout << squareRoot(value);
} catch (invalid_argument &e) {
    cerr << "cannot do that: " << e.what() << endl;
}
```

Needs `#include <stdexcept>`. Catch by **reference**: catching by value slices the exception object.

TALK TO YOUR NEIGHBOR · TTYN

What does this program print?

```
void f() { throw invalid_argument("yuck");
           cout << "a"; }
void g() { cout << "b"; f(); cout << "c"; }

int main() { cout << "d"; g(); f(); cout << "e"; }
```

A. dbe

B. dbc

C. dbace

D. dbae

E. None of the others

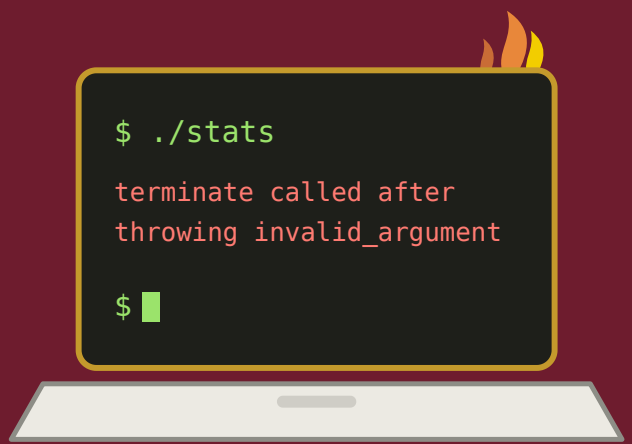
Stack Unwinding

A `throw` behaves like a `return` that keeps going until someone catches it.

`main()`	has try/catch → handles it ✓
`g()`	no handler → frame destroyed, keep going ↑
`f()`	throw starts here ↑

Every local object in each abandoned frame gets its **destructor called** on the way out. If nobody catches it, the program terminates.

Demo



Reading a file that isn't there.

(the file was right there. in the other folder.)

Two-Dimensional Arrays

```
const unsigned ROWS = 5;
const unsigned COLS = 8;

char board[ROWS][COLS];          // 5 rows, 8 columns

for (unsigned r = 0; r < ROWS; ++r) {
    for (unsigned c = 0; c < COLS; ++c) {
        board[r][c] = '.';
    }
}
```

Row first, then column: `board[r][c]`. The outer loop walks rows, the inner walks the cells within a row.

How 2D Arrays Really Sit in Memory

`int grid[2][4]` is not a grid; it is 8 ints in a row.

row 0	[0][0]	[0][1]	[0][2]	[0][3]
row 1	[1][0]	[1][1]	[1][2]	[1][3]
in memory	one contiguous block, row after row			

This is why a pointer walking the array crosses from the end of one row into the start of the next, and why the compiler needs the column count to do the arithmetic.

Passing a 2D Array

```
const unsigned COLS = 8;

void printBoard(char b[][COLS], unsigned rows) {
    ...
}
```

- You may leave the **row** count off: pass it as a separate argument
- You may **not** leave the column count off
- Without the columns, `b[r][c]` can't be turned into an address

> Same rule as 1D arrays, one dimension up: the function receives an address, so it needs the shape spelled out.

Week 02 Recap

- File streams use the **same operators** as `cin`/`cout`
- Always check `is_open()`: a missing file fails silently
- **Read, then test**: `while (getline(fin, line))`
- `throw` hands the problem to whoever can decide what to do
- An uncaught exception unwinds every frame, then terminates
- 2D arrays are one contiguous block, the column count is not optional

Next: a03 →

This deck stands on earlier CS112 materials by
Joel Adams and **Victor Norman** · adapted and extended by **Eric Araújo**