

C++

File Writing & Reading

*originally developed by Weiguang Cui,
refined by Alexander Knebe*



ofstream



ifstream



fstream



Modes



Status



Binary

Input / Output with Files — <fstream>

ofstream

Stream class to write to files

ifstream

Stream class to read from files

fstream

Stream class to read and write from/to files



```
#include <fstream>
using namespace std;

int main () {
    ofstream myfile;                // declare the file stream variable
    myfile.open("example.txt");      // open (create) the file
    myfile << "Writing this to a file.\n"; // works just like cout
    myfile.close();                  // always close when done!
    return 0;
}
```

Always call `.close()` when finished. If you forget, buffered data may not be flushed and the file handle is not released until the program exits.

File Open Modes

```
myfile.open("filename", mode); // mode is optional
```

ios::in

Open for input (reading).

ios::out

Open for output (writing).

ios::binary

Open in binary mode. Omitting this gives a plain text file.

ios::ate

Set the initial position to the end of the file. Without this, starts at the beginning.

ios::app

All output is appended to the existing content — nothing is overwritten.

ios::trunc

If the file exists and is opened for writing, its previous content is deleted first.

Modes can be combined with | e.g. `myfile.open("ex.bin", ios::out | ios::app | ios::binary)`

Default Modes & Cross-Mode Behaviour

Default mode per class

Class	Default Mode
<code>ofstream</code>	<code>ios::out</code>
<code>ifstream</code>	<code>ios::in</code>
<code>fstream</code>	<code>ios::in ios::out</code>

Try it!

Can you open with `ios::in` on an `ofstream`?

→ `ofstream myfile;`
`myfile.open("file.txt", ios::in);`

Can you open with `ios::out` on an `ifstream`?

→ What happens if the file already has content?

Run each experiment with a file that already exists and observe what changes.

What to look for:

- `ofstream` with `ios::in` — the file is still opened for writing by default; adding `ios::in` gives read access too, but behaviour depends on whether the file exists.
- `ifstream` with `ios::out` — read-only streams cannot write; this combination will typically fail or be ignored.
- The key insight: each class has a sensible default, but you can override it. Always test with an existing file to see what changes.

Checking File Status



```
if (myfile.is_open()) { /* ok — proceed */ }
```

`is_open()`

Returns true if the file was successfully opened. Always check this before reading or writing.

`eof()`

Returns true when a file opened for reading has reached the end of the file.

`bad()`

Returns true when a read/write operation fails — e.g. writing to a read-only file, or the device is full.

`good()`

Returns false if any of the above flags are set. Note: `good()` and `bad()` are NOT exact opposites — `good()` checks more flags.

`fail()`

Same as `bad()`, plus format errors — e.g. trying to read an integer but finding a letter.

Practice — Writing to a File

1

Write 1–100

Write the numbers 1 to 100 into a text file, one per line. Open the file afterward and verify.

2

Two columns

Write two numbers per line — e.g. i and $i*i$ — separated by a space or tab. Check that reading it back is easy.

3

Separator test

Try `myfile << a << b` with no separator. What does the file look like? How do you fix it so values are clearly separated?

4

Precision

Control output precision:

```
myfile.precision(5);  
myfile << setprecision(5) << 1000.243;  
(Requires #include <iomanip>)
```

Reading from a File



```
#include <iostream>
#include <fstream>
#include <string>
using namespace std;

int main () {
    string line;
    ifstream myfile("example.txt");

    if (myfile.is_open()) {
        while (getline(myfile, line)) { // reads one full
line at a time
            cout << line << '\n';
        }
        myfile.close();
    } else {
        cout << "Unable to open file";
    }
    return 0;
}
```

getline vs >>

getline(f, line)

Reads until `\n` — captures the entire line including spaces.

myfile >> word

Reads until whitespace — good for word-by-word or number-by-number reading.

eof() loop

When using `>>`, loop with `while (!myfile.eof())` or `while (myfile >> val)` — not needed with `getline`.

Always check

Call `is_open()` before reading. A missing file gives no exception by default — just a silent failure.

Try replacing `getline` with `myfile >> line`. What changes? You will need `eof()` in the while condition.

Write & Read a Matrix

Can you directly write a matrix with `myfile << mymatrix` and read it back with `myfile >> mymatrix`? Try it!

Writing a matrix

```
// Using Eigen library:
MatrixXd M(2,3);
M << 1,2,3, 4,5,6;
ofstream f("matrix.txt");
f << M;
f.close();
```

Eigen overloads `<<` for file output — rows separated by newlines, values by spaces.

Reading a matrix

```
// Manual row-by-row read:
ifstream f("matrix.txt");
for(int i=0;i<rows;i++)
    for(int j=0;j<cols;j++)
        f >> M(i,j);
f.close();
```

Eigen does NOT overload `>>` for file input — you must read element by element.

Practice — Write & Read a Data File



```
#include <fstream>
#include <cmath>      // sin()
#include <cstdlib>     // rand()
using namespace std;

int main() {
    ofstream f("data.txt");
    f.precision(6);
    for (double x = 0.0; x <= 5.0; x += 0.1) {
        double noise = (rand() % 200 - 100) / 1000.0; //
±0.1
        double y = 3.0 * sin(x) + noise;
        f << x << "\t" << y << "\n"; // tab-separated
    }
    f.close();

    // Read back:
    double xv, yv;
    ifstream fin("data.txt");
    while (fin >> xv >> yv)
        cout << xv << "\t" << yv << "\n";
    fin.close();
}
```

Tasks

Write x from 0 to 5 (step 0.1) and $y = 3 \cdot \sin(x) + \text{noise}$ into a two-column text file.

Read the file back and print both columns to verify your write worked correctly.

Extension: try writing the same data in binary format (`ios::binary`). Open the binary file in a text editor — what do you see?

What are the key differences between text and binary files in terms of size, portability, and readability?

Text files: human-readable, larger, portable. Binary files: machine-readable, compact, platform-dependent byte order.