File Operations in C++

In C++, file operations are typically performed using the `<fstream>` library, which provides classes for reading from and writing to files. Here are the most commonly used file operations:

1. Including the Necessary Header

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

2. Opening a File

- Using `std::ifstream` for input (reading from a file).
- Using `std::ofstream` for output (writing to a file).
- Using `std::fstream` for both input and output.

```
std::ifstream inFile("input.txt"); // Open for reading
std::ofstream outFile("output.txt"); // Open for writing
std::fstream ioFile("data.txt", std::ios::in |
    std::ios::out); // Open for both
```

3. Checking if a File is Open

```
if (!inFile.is_open()) {
    std::cerr << "Error opening file!" << std::endl;
    return 1;
}
```

4. Reading from a File

```
std::string line;
while (std::getline(inFile, line)) {
    std::cout << line << std::endl;
}
```

```cpp
std::string word;
while (inFile >> word) {
    std::cout << word << std::endl;
}
```

```cpp
char ch;
while (inFile.get(ch)) {
    std::cout << ch;
}
```

5. Writing to a File

```cpp
outFile << "Hello, World!" << std::endl;
outFile << "This is a new line." << std::endl;
```

```cpp
int num = 42;
double pi = 3.14159;
outFile << "Number: " << num << ", Pi: " << pi << std::endl;
```

6. Closing a File

```cpp
inFile.close();
outFile.close();
ioFile.close();
```

7. Appending to a File

```cpp
std::ofstream appFile("output.txt", std::ios::app); // Open
    in append mode
appFile << "This will be appended." << std::endl;
appFile.close();
```

8. Checking for End of File (EOF)

```
while (!inFile.eof()) {
    // Read data
}
```

9. File Positioning (Seek & Tell)

- `tellg()` / `tellp()` – Get current position in file.
- `seekg()` / `seekp()` – Set position in file.

```
inFile.seekg(0, std::ios::beg); // Move to the beginning
std::streampos pos = inFile.tellg(); // Get current position
```

10. Error Handling

```
if (inFile.fail()) {
    std::cerr << "Input operation failed!" << std::endl;
}
if (outFile.bad()) {
    std::cerr << "Critical error while writing!" <<
        std::endl;
}
```

Complete Example: Reading & Writing a File

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

int main() {
    // Writing to a file
    std::ofstream outFile("example.txt");
    if (!outFile) {
        std::cerr << "Could not open file for writing!" <<
            std::endl;
        return 1;
    }
    outFile << "Hello, File!\n";
```

```
13      outFile << "This is line 2.\n";
14      outFile.close();
15
16      // Reading from a file
17      std::ifstream inFile("example.txt");
18      if (!inFile) {
19          std::cerr << "Could not open file for reading!" <<
                std::endl;
20          return 1;
21      }
22      std::string line;
23      while (std::getline(inFile, line)) {
24          std::cout << line << std::endl;
25      }
26      inFile.close();
27
28      return 0;
29  }
```

Common File Open Modes

Mode	Description
`std::ios::in`	Open for reading
`std::ios::out`	Open for writing (truncates if exists)
`std::ios::app`	Append mode (write at end)
`std::ios::ate`	Open and seek to end
`std::ios::binary`	Open in binary mode
`std::ios::trunc`	Truncate file if it exists

File Open Modes in C++

In C++, file operations are performed using file streams from the `<fstream>` library. The file open modes are specified using `std::ios` flags, which control how a file is opened and manipulated. Below is a detailed explanation of the most commonly used file open modes in C++:

---

1. `std::ios::in` (Open for Reading)

- Opens a file for **input (reading)**.
- The file must exist; otherwise, opening fails.
- Used with `ifstream` (input file stream).

Example:

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

int main() {
    std::ifstream inFile("input.txt", std::ios::in);
    if (!inFile) {
        std::cerr << "Error opening file for reading!" <<
            std::endl;
        return 1;
    }
    // Read from file...
    inFile.close();
    return 0;
}
```

---

2. `std::ios::out` (Open for Writing)

- Opens a file for **output (writing)**.
- If the file exists, it is **truncated (cleared)** by default.
- If the file does not exist, it is **created**.
- Used with `ofstream` (output file stream).

Example:

```
std::ofstream outFile("output.txt", std::ios::out);
if (!outFile) {
    std::cerr << "Error opening file for writing!" <<
        std::endl;
    return 1;
}
outFile << "This will overwrite the file." << std::endl;
outFile.close();
```

---

3. `std::ios::app` (Append Mode)

- Opens a file for writing, but new data is **appended** to the end.
- Does **not truncate** the existing content.
- If the file does not exist, it is created.

Example:

```
std::ofstream appFile("log.txt", std::ios::app);
appFile << "This line is appended." << std::endl;
appFile.close();
```

---

4. `std::ios::ate` (Open and Seek to End)

- Opens the file and **moves the pointer to the end**.
- Allows reading/writing anywhere in the file (unlike `app`, which only allows appending).
- Does **not truncate** the file.

Example:

```
std::fstream file("data.txt", std::ios::in | std::ios::out
    | std::ios::ate);
if (!file) {
    std::cerr << "Error opening file!" << std::endl;
    return 1;
}
file << "Added at the end." << std::endl;
file.seekg(0); // Move back to the start for reading
file.close();
```

---

5. `std::ios::binary` (Binary Mode)

- Opens the file in **binary mode** (no text formatting).
- Used for non-text files (images, executables, etc.).
- Avoids automatic newline conversions (important on Windows).

Example:

```cpp
std::ifstream binFile("image.png", std::ios::binary);
if (!binFile) {
    std::cerr << "Error opening binary file!" << std::endl;
    return 1;
}
// Read raw binary data...
binFile.close();
```

---

6. `std::ios::trunc` (Truncate Mode)

- **Deletes the existing content** when opening.
- Often used with `std::ios::out` (default behavior of `ofstream`).
- If the file does not exist, it is created.

Example:

```cpp
std::ofstream truncFile("temp.txt", std::ios::out |
    std::ios::trunc);
truncFile << "This replaces all existing content." <<
    std::endl;
truncFile.close();
```

Combining Modes

Modes can be combined using the bitwise OR (`|`) operator.

Example: Open for both reading and writing

```cpp
std::fstream rwFile("data.txt", std::ios::in |
    std::ios::out);
if (!rwFile) {
    std::cerr << "Error opening file for read/write!" <<
        std::endl;
    return 1;
}
rwFile << "Writing..." << std::endl;
rwFile.seekg(0); // Move to start for reading
std::string line;
std::getline(rwFile, line);
std::cout << "Read: " << line << std::endl;
rwFile.close();
```

Default Modes

Class	Default Mode	Behavior
`std::ifstream`	`std::ios::in`	Open for reading
`std::ofstream`	`std::ios::out`	Open for writing (truncates)
`std::fstream`	None	Must specify modes explicitly

Common Use Cases

Operation	Recommended Mode
Read a text file	`std::ios::in`
Write a new file (overwrite)	`std::ios::out` (or `std::ios::out | std::ios::trunc`)
Append to a file	`std::ios::app`
Read + Write	`std::ios::in | std::ios::out`
Binary file I/O	`std::ios::binary`

Error Handling

Always check if the file opened successfully:

```
std::ifstream file("missing.txt");
if (!file.is_open()) {
    std::cerr << "Failed to open file!" << std::endl;
    return 1;
}
```

Summary Table of Modes

Mode	Description	Commonly Used With
`std::ios::in`	Read mode	`ifstream`
`std::ios::out`	Write mode (truncates)	`ofstream`
`std::ios::app`	Append mode	`ofstream`
`std::ios::ate`	Start at end	`fstream`
`std::ios::binary`	Binary mode	All streams
`std::ios::trunc`	Truncate on open	`ofstream`

Python File Operations

File operations in Python are straightforward and commonly performed using built-in functions. Below are the most frequently used file operations:

---

1. Opening a File

Use the `open()` function to open a file. It returns a file object.

```
file = open("filename.txt", "mode")  # Specify filename and
    mode
```

Common File Modes:

Mode	Description
`"r"`	Read (default)
`"w"`	Write (creates new or truncates existing)
`"a"`	Append (writes at the end)
`"r+"`	Read + Write
`"rb"` / `"wb"`	Binary read/write
`"x"`	Exclusive creation (fails if file exists)

```
file = open("example.txt", "r")  # Open for reading
```

---

2. Reading from a File

```
content = file.read()  # Reads entire content as a string
print(content)
```

```
for line in file:
    print(line.strip())  # strip() removes extra newlines
```

```
lines = file.readlines()  # Returns a list of lines
print(lines)
```

```
first_line = file.readline()  # Reads one line
print(first_line)
```

---

3. Writing to a File

```
file = open("output.txt", "w")
file.write("Hello, World!\n")  # Writes a line
file.close()
```

```
lines = ["Line 1\n", "Line 2\n", "Line 3\n"]
file.writelines(lines)  # Writes a list of strings
file.close()
```

---

4. Appending to a File

```
file = open("output.txt", "a")
file.write("This is appended text.\n")
file.close()
```

---

5. Closing a File

Always close files to free resources.

```
file = open("example.txt", "r")
# ... do operations ...
file.close()  # Important!
```

Automatic Closing (Recommended): Using `with` ensures the file is closed automatically.

```
with open("example.txt", "r") as file:
    content = file.read()
    # No need to close explicitly
```

---

6. Checking if a File Exists

Use `os.path.exists()`:

```
import os
if os.path.exists("example.txt"):
    print("File exists!")
else:
    print("File not found.")
```

---

7. File Position Handling

- `tell()` → Returns current file pointer position.
- `seek(offset, whence)` → Moves file pointer.

```
with open("example.txt", "r") as file:
    print(file.tell())  # Current position (0 at start)
    file.seek(10)       # Move to 10th byte
    print(file.read(5)) # Read next 5 bytes
```

---

8. Binary File Operations

Read/write binary data (e.g., images, PDFs):

```
with open("image.png", "rb") as file:
    data = file.read()

with open("copy.png", "wb") as file:
    file.write(data)
```

---

9. Common File Operations (Using `os` and `shutil`)

```
import os
os.remove("file.txt")
```

```
os.rename("old.txt", "new.txt")
```

```python
import shutil
shutil.copy("source.txt", "destination.txt")
```

```python
size = os.path.getsize("example.txt")
print(f"Size: {size} bytes")
```

---

10. Error Handling (Try-Except)

```python
try:
    with open("nonexistent.txt", "r") as file:
        print(file.read())
except FileNotFoundError:
    print("File not found!")
except IOError as e:
    print(f"An error occurred: {e}")
```

Full Example: Read & Write

```python
# Writing
with open("test.txt", "w") as file:
    file.write("Hello, Python!\n")
    file.write("This is line 2.\n")

# Reading
with open("test.txt", "r") as file:
    for line in file:
        print(line.strip())
```

Summary of Most Common Operations

Operation	**Method**
Open	`open("file.txt", "r")`
Read all	`file.read()`
Read lines	`file.readlines()`
Write	`file.write("text")`
Append	`open("file.txt", "a")`
Close	`file.close()` or `with`
Check existence	`os.path.exists()`

Python's file handling is simple and powerful, especially when using `with` for automatic resource management.