Gorgo Continuing C++

April 11, 2026

Contents

11. The Filesystem Library

In *Gorgo Starting C++* you learned to read and write file contents with `std::fstream`. But working with files means more than reading and writing: you often need to check whether a file exists, list directory contents, copy or rename files, or build paths that work across operating systems. Before C++17, you needed platform-specific APIs (POSIX or Windows) for all of this. C++17 introduced `<filesystem>`, a portable library for working with paths, directories, and file metadata. In this chapter you will learn `std::filesystem::path`, directory iteration, file operations, and file status queries.

All filesystem types and functions live in the `std::filesystem` namespace. We will use the common alias:

```
#include <filesystem>
namespace fs = std::filesystem;
```

`std::filesystem::path`

A `fs::path` represents a file path in a platform-independent way. It handles separator differences (`/` on Unix, `\` on Windows) automatically:

```
#include <filesystem>
#include <iostream>

namespace fs = std::filesystem;

int main()
{
    fs::path p = "/home/user/music/playlist.m3u";

    std::cout << "Full path:  " << p << "\n";
    std::cout << "Filename:   " << p.filename() << "\n";
    std::cout << "Stem:       " << p.stem() << "\n";
    std::cout << "Extension:  " << p.extension() << "\n";
    std::cout << "Parent:     " << p.parent_path() << "\n";

    return 0;
}
```

```
Full path:  "/home/user/music/playlist.m3u"
Filename:   "playlist.m3u"
Stem:       "playlist"
Extension:  ".m3u"
Parent:     "/home/user/music"
```

Building Paths with /

The `/` operator concatenates paths, inserting the correct separator:

```
fs::path base = "/home/user";
fs::path full = base / "music" / "album" / "track01.mp3";
std::cout << full << "\n";
// "/home/user/music/album/track01.mp3"
```

This is much cleaner than string concatenation and handles platform differences automatically.

Other Path Operations

```
fs::path p = "/home/user/docs/../music/./track.mp3";

// Normalize the path (resolve . and ..)
std::cout << p.lexically_normal() << "\n";
// "/home/user/music/track.mp3"

// Make relative to another path
fs::path base = "/home/user";
std::cout << p.lexically_relative(base) << "\n";
// "docs/../music/./track.mp3"

// Check if path is absolute or relative
std::cout << p.is_absolute() << "\n";  // true
std::cout << fs::path("music/track.mp3").is_relative() << "\n";  // true
```

Checking File Status

Before operating on a file, you often need to check whether it exists and what kind of entry it is:

```
fs::path p = "/home/user/music";

// bool exists(const path& p);
if (fs::exists(p)) {
    std::cout << p << " exists\n";
}

// bool is_regular_file(const path& p);
// bool is_directory(const path& p);
// bool is_symlink(const path& p);
if (fs::is_directory(p)) {
    std::cout << p << " is a directory\n";
}

if (fs::is_regular_file(p / "track.mp3")) {
    std::cout << "It's a file\n";
}
```

File Size

```
// uintmax_t file_size(const path& p);
auto size = fs::file_size("/home/user/music/track.mp3");
std::cout << "Size: " << size << " bytes\n";
```

Trap: `file_size` throws `filesystem_error` if the file does not exist or you do not have permission. Check `exists()` first, or use the overload that takes a `std::error_code` parameter.

Error Handling

Most filesystem functions have two overloads: - One that throws `fs::filesystem_error` on failure. - One that takes a `std::error_code&` parameter and sets it instead of throwing.

```cpp
std::error_code ec;
auto size = fs::file_size("nonexistent.txt", ec);
if (ec) {
    std::cout << "Error: " << ec.message() << "\n";
} else {
    std::cout << "Size: " << size << "\n";
}
```

Directory Iteration

`directory_iterator`

`fs::directory_iterator` lists the entries in a single directory:

```cpp
#include <filesystem>
#include <iostream>

namespace fs = std::filesystem;

int main()
{
    fs::path dir = "/home/user/music";

    for (const auto& entry : fs::directory_iterator(dir)) {
        std::cout << entry.path().filename();
        if (entry.is_directory()) {
            std::cout << "/";
        }
        std::cout << "\n";
    }

    return 0;
}
```

Each `entry` is a `fs::directory_entry` with methods like `path()`, `is_regular_file()`, `is_directory()`, and `file_size()`.

`recursive_directory_iterator`

`fs::recursive_directory_iterator` walks the entire directory tree:

```cpp
for (const auto& entry : fs::recursive_directory_iterator(dir)) {
    if (entry.is_regular_file() && entry.path().extension() == ".mp3") {
        std::cout << entry.path() << "\n";
    }
}
```

Tip: `recursive_directory_iterator` follows symlinks by default on some platforms. Use `fs::directory_options::skip_permission_denied` to avoid exceptions on directories you cannot read:

```cpp
for (const auto& entry : fs::recursive_directory_iterator(
        dir, fs::directory_options::skip_permission_denied)) {
    // ...
}
```

File Operations

Creating Directories

```
// bool create_directory(const path& p);
fs::create_directory("/home/user/music/new_album");

// bool create_directories(const path& p);  — creates parent dirs too
fs::create_directories("/home/user/music/2005/singles");
```

`create_directory` fails if the parent does not exist. `create_directories` creates the entire path.

Copying Files

```
// void copy(const path& from, const path& to);
// void copy_file(const path& from, const path& to);
fs::copy_file("track01.mp3", "backup/track01.mp3");

// With options
fs::copy_file("track01.mp3", "backup/track01.mp3",
              fs::copy_options::overwrite_existing);
```

Copy options:

Option	Effect
`none`	Fail if destination exists (default)
`overwrite_existing`	Replace the destination
`skip_existing`	Silently skip if destination exists
`update_existing`	Replace only if source is newer

Renaming and Moving

```
// void rename(const path& old_p, const path& new_p);
fs::rename("track01.mp3", "01_intro.mp3");
fs::rename("old_dir", "new_dir");  // works for directories too
```

`rename` can also move files across directories on the same filesystem.

Removing Files

```
// bool remove(const path& p);
fs::remove("temp.txt");

// uintmax_t remove_all(const path& p);  — removes directory and all contents
auto count = fs::remove_all("old_backup");
std::cout << "Removed " << count << " entries\n";
```

Trap: `remove_all` recursively deletes everything inside a directory. Double-check the path before calling it — there is no "undo."

File Permissions

You can query and modify file permissions:

```cpp
// fs::perms permissions(const path& p);
auto perms = fs::status("script.sh").permissions();

if ((perms & fs::perms::owner_exec) != fs::perms::none) {
    std::cout << "Owner can execute\n";
}

// void permissions(const path& p, perms prms, perm_options opts);
fs::permissions("script.sh",
    fs::perms::owner_exec,
    fs::perm_options::add);
```

Common permission flags:

Flag	Meaning
`owner_read`	Owner can read
`owner_write`	Owner can write
`owner_exec`	Owner can execute
`group_read`	Group can read
`others_read`	Others can read
`all`	All permissions

Try It: File Manager

Here is a program that exercises the filesystem library. Type it in, compile with `g++ -std=c++23`, and experiment:

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

namespace fs = std::filesystem;

void create_test_files(const fs::path& dir)
{
    fs::create_directories(dir / "rock");
    fs::create_directories(dir / "pop");

    std::ofstream(dir / "rock" / "boulevard.txt") << "Boulevard of Broken Dreams\n";
    std::ofstream(dir / "rock" / "numb.txt") << "Numb\n";
    std::ofstream(dir / "pop" / "umbrella.txt") << "Umbrella\n";
    std::ofstream(dir / "readme.txt") << "Test files\n";
}

int main()
{
    fs::path test_dir = fs::temp_directory_path() / "fs_test";

    // Clean up from previous runs
    fs::remove_all(test_dir);

    // Create test structure
    create_test_files(test_dir);
```

```
    std::cout << "Created files in: " << test_dir << "\n\n";

    // List all files recursively
    std::cout << "All files:\n";
    for (const auto& entry : fs::recursive_directory_iterator(test_dir)) {
        auto rel = fs::relative(entry.path(), test_dir);
        if (entry.is_regular_file()) {
            std::cout << "  " << rel << " (" << entry.file_size() << " bytes)\n";
        } else if (entry.is_directory()) {
            std::cout << "  " << rel << "/\n";
        }
    }

    // Copy a file
    fs::copy_file(test_dir / "rock" / "numb.txt",
                  test_dir / "pop" / "numb_copy.txt");
    std::cout << "\nCopied numb.txt to pop/\n";

    // Rename
    fs::rename(test_dir / "readme.txt", test_dir / "README.txt");
    std::cout << "Renamed readme.txt -> README.txt\n";

    // Check existence
    std::cout << "\nREADME.txt exists: "
              << fs::exists(test_dir / "README.txt") << "\n";
    std::cout << "readme.txt exists: "
              << fs::exists(test_dir / "readme.txt") << "\n";

    // Count .txt files
    int txt_count = 0;
    for (const auto& entry : fs::recursive_directory_iterator(test_dir)) {
        if (entry.path().extension() == ".txt") {
            txt_count++;
        }
    }
    std::cout << "\nTotal .txt files: " << txt_count << "\n";

    // Cleanup
    auto removed = fs::remove_all(test_dir);
    std::cout << "Cleaned up " << removed << " entries\n";

    return 0;
}
```

Try modifying this to find the largest file, list only directories, or filter by extension.

Key Points

- **`std::filesystem::path`** represents file paths portably. Use `/` operator to concatenate paths.
- `filename()`, `stem()`, `extension()`, and `parent_path()` decompose paths.
- `lexically_normal()` resolves `.` and `..` segments.
- **`exists()`**, **`is_regular_file()`**, **`is_directory()`** check file status.
- **`file_size()`** returns the size of a regular file in bytes.
- **`directory_iterator`** lists a single directory; **`recursive_directory_iterator`** walks the entire tree.

- **`create_directory`** creates one directory; **`create_directories`** creates the entire path.
- **`copy_file`** copies a file; use `copy_options` to control overwrite behavior.
- **`rename`** renames or moves entries; **`remove`** deletes a file; **`remove_all`** deletes a directory tree.
- Most functions have a throwing overload and a `std::error_code` overload for error handling.
- File permissions can be queried with `status().permissions()` and modified with `permissions()`.

Exercises

1. **Think about it:** Why does `std::filesystem::path` use `/` as the concatenation operator instead of `+`? What would go wrong with `+`?

2. **What does this print?**

```cpp
fs::path p = "/home/user/docs/report.pdf";
std::cout << p.stem() << "\n";
std::cout << p.extension() << "\n";
std::cout << p.parent_path().filename() << "\n";
```

3. **Where is the bug?**

```cpp
auto size = fs::file_size("maybe_missing.txt");
std::cout << "Size: " << size << "\n";
```

4. **Think about it:** What is the difference between `directory_iterator` and `recursive_directory_iterator`? When would you use each?

5. **What does this print?**

```cpp
fs::path p = "/home/user/./docs/../music/track.mp3";
std::cout << p.lexically_normal() << "\n";
```

6. **Calculation:** You call `create_directories("/a/b/c/d")` on a system where only `/a` exists. How many new directories are created?

7. **Where is the bug?**

```cpp
fs::create_directory("/new_project/src/main");
```

   (Assume `/new_project` does not exist yet.)

8. **Think about it:** Why does `remove_all` return the number of entries removed? When would this be useful?

9. **What happens?**

```cpp
fs::copy_file("a.txt", "b.txt");
fs::copy_file("a.txt", "b.txt");
```

   What happens on the second call? How would you fix it?

10. **Write a program** that takes a directory path as a command-line argument and prints a summary: the total number of files, the total number of directories, and the total size of all regular files in bytes. Use `recursive_directory_iterator` to walk the tree.