Examples of using std:filesystem

In this section, we'll analyze a few examples where `std::filesystem` is used. We'll go from a simple code - loading a file into a string, and then explore creating directories, or filtering using `std::regex`.

WE'LL COVER THE FOLLOWING

To revise what we have learned so far, here is the sample code that lists the files in the directories and displays their size and when were they last modified:

input.cpp

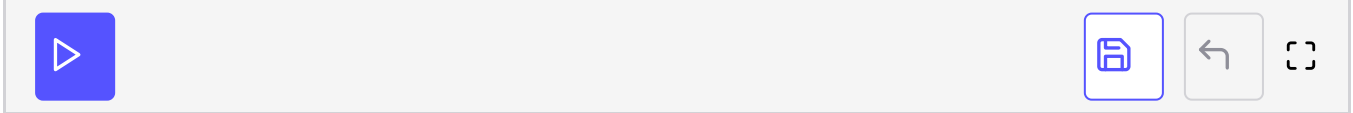
test1.txt

test2.txt

```
#include <chrono>
#include <filesystem>
#include <iostream>
#include <iomanip>
#include <optional>
#include <sstream>
#include <string>
#ifdef _MSC_VER
#define NOMINMAX
#include <windows.h>
#endif

namespace fs = std::filesystem;

std::optional<std::uintmax_t> ComputeFileSize(const fs::path& pathToCheck)
{
    if (fs::exists(pathToCheck) && fs::is_regular_file(pathToCheck))
    {
        auto err = std::error_code{};
        const auto filesize = fs::file_size(pathToCheck, err);
```

```
        const auto filesize = fs::file_size(pathToCheck, err);
        if (err == std::error_code{} && filesize != static_cast<uintmax_t>(-1))
            return filesize;
    }

    return std::nullopt;
}

template <typename UnitStr, typename ... UnitsStr>
std::string UnitString(double value, double, UnitStr str)
{
    return std::to_string(value) + ' ' + str;
}

template <typename UnitStr, typename ... UnitsStr>
std::string UnitString(double value, double unitStep, UnitStr str, UnitsStr ... strs)
{
    if (value > unitStep)
        return UnitString(value / unitStep, unitStep, strs...);
    else
        return UnitString(value, unitStep, str);
}

std::string SizeToString(std::optional<std::uintmax_t> fsize)
{
    if (fsize)
        return UnitString(static_cast<double>(*fsize), 1024, "B", "KB", "MB", "GB");

    return "err";
}

template <typename TDuration>
std::string ElapsedToString(const TDuration& dur)
{
    const auto elapsed = std::chrono::duration_cast<std::chrono::seconds>(dur).count();
    return UnitString(static_cast<double>(elapsed), 60.0, "seconds", "minutes", "hours");
}

std::string FileTimeToDate(fs::file_time_type t)
{
#ifdef _MSC_VER
    FILETIME ft;
    memcpy(&ft, &t, sizeof(FILETIME));
    SYSTEMTIME stSystemTime;
    if (FileTimeToSystemTime(&ft, &stSystemTime)) {
        return std::to_string(stSystemTime.wDay) + "/" +
            std::to_string(stSystemTime.wMonth) + "/" +
            std::to_string(stSystemTime.wYear) + " " +
            std::to_string(stSystemTime.wHour) + ":" +
            std::to_string(stSystemTime.wMinute) + ":" +
            std::to_string(stSystemTime.wSecond);
    }
    return "";
#else

    auto convFileTime = std::chrono::time_point_cast<std::chrono::system_clock::duration>(t
    std::time_t convfiletime = std::chrono::system_clock::to_time_t(convFileTime);

    return std::ctime(&convfiletime);
#endif
}

void DisplayFileInfo(const fs::directory_entry & entry, int level)
```

```
{
    const auto filetime = fs::last_write_time(entry);
    const auto ofsize = ComputeFileSize(entry);

    std::cout << std::setw(level * 3) << " " << entry.path().filename() << ", "
        << SizeToString(ofsize)
        << ", modified: "
        << ElapsedToString(fs::file_time_type::clock::now() - filetime)
        << ", date: "
        << FileTimeToDate(filetime)
        << '\n';
}

void DisplayDirectoryTree(const fs::path& pathToShow, int level = 0)
{
    if (fs::exists(pathToShow) && fs::is_directory(pathToShow))
    {
        for (const auto& entry : fs::directory_iterator(pathToShow))
        {
            auto filename = entry.path().filename();
            if (fs::is_directory(entry.status()))
            {
                std::cout << std::setw(level * 3) << "" << "[+] " << filename << '\n';
                DisplayDirectoryTree(entry, level + 1);
            }
            else if (fs::is_regular_file(entry.status()))
                DisplayFileInfo(entry, level);
            else
                std::cout << std::setw(level * 3) << "" << " [?]" << filename << '\n';
        }
    }
}

int main(int argc, char* argv[])
{
    try
    {
        const fs::path pathToShow{ argc >= 2 ? argv[1] : fs::current_path() };

        std::cout << "listing files in the directory: " << fs::absolute(pathToShow) << '\n';
        std::cout << "current path is: " << fs::current_path() << '\n';

        DisplayDirectoryTree(pathToShow);
    }
    catch (const fs::filesystem_error& err)
    {
        std::cerr << "filesystem error! " << err.what() << '\n';
    }
    catch (...)
    {
        std::cerr << "unknown exception!\n";
    }
}
```



For more use cases you can also read another chapter – How to Parallelise CSV Reader - where `std::filesystem` is a crucial element for finding CSV files.

Loading a File into a String

The first example shows a compelling case where we leverage `std::filesystem`'s size related functions to build a buffer for the file contents.

Here's the code compiled in GCC using:

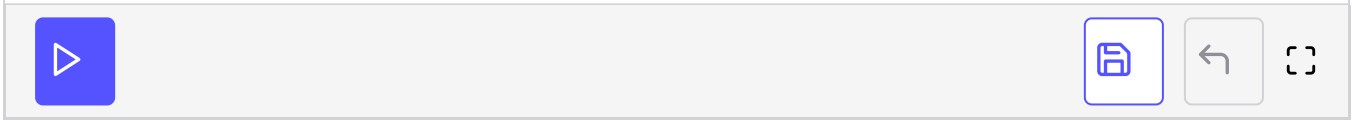
```
g++ -std=c++17 input.cpp -lstdc++fs
./a.out "./text.txt"
```

input.cpp

text.txt

```
#include <chrono>
#include <filesystem>
#include <iostream>
#include <fstream>

namespace fs = std::filesystem;

[[nodiscard]] std::string GetFileContents(const fs::path& filePath)
{
    std::ifstream inFile{ filePath, std::ios::in | std::ios::binary };
    if (!inFile)
        throw std::runtime_error("Cannot open " + filePath.filename().string());

    const auto fsize = fs::file_size(filePath);
    if (fsize > std::numeric_limits<size_t>::max())
        throw std::runtime_error("file is too large to fit into size_t! " + filePath.filenam

    std::string str(static_cast<size_t>(fsize), 0);

    inFile.read(str.data(), str.size());
    if (!inFile)
        throw std::runtime_error("Could not read the full contents from " + filePath.filenam

    return str;
}

int main(int argc, char* argv[])
{
    try
    {
        if (argc > 1)
        {
            const auto str = GetFileContents(argv[1]);
            std::cout << "string size: " << str.size() << '\n';
        }
    }
    catch (const fs::filesystem_error& err)
```

```
    {
        std::cerr << "filesystem error! " << err.what() << '\n';
        if (!err.path1().empty())
            std::cerr << "path1: " << err.path1().string() << '\n';
        if (!err.path2().empty())
            std::cerr << "path2: " << err.path2().string() << '\n';
    }
    catch (const std::exception& ex)
    {
        std::cerr << "general exception: " << ex.what() << '\n';
    }
}
```



Before C++17 to get the file size you'd usually reposition the file pointer to the end and then read the position again. For example:

```
ifstream testFile("test.file", ios::binary);
const  auto begin = myfile.tellg();
testFile.seekg (0, ios::end);
const auto end = testFile.tellg();
const auto fsize =  (end-begin);
```

You can also open a file with `ios::ate` flag and then the file pointer will be positioned automatically at the end.

However, all of the above methods require to open a file but with `std::filesystem` the code is much shorter.

Creating Directories

In the second example, we'll build `N` directories each with `M` files.

The core part of the `main()`:

```
const fs::path startingPath{ argc >= 2 ? argv[1] : fs::current_path() };
const std::string strTempName{ argc >= 3 ? argv[2] : "temp" };
const int numDirectories{ argc >= 4 ? std::stoi(argv[3]) : 4 };
const int numFiles{ argc >= 5 ? std::stoi(argv[4]) : 4 };

if (numDirectories < 0 || numFiles < 0)
    throw std::runtime_error("negative input numbers...");

const fs::path tempPath = startingPath / strTempName;
CreateTempData(tempPath, numDirectories, numFiles);
```

And the `CreateTempData()` function:

```
std::vector<fs::path> GeneratePathNames(const fs::path& tempPath,unsigned num)
{
    std::vector<fs::path> outPaths{ num, tempPath };
    for (auto& dirName : outPaths)
    {
        // use pointer value to generate unique name...
        const auto addr = reinterpret_cast<uintptr_t>(&dirName);
        dirName /= std::string("tt") + std::to_string(addr);
    }
    return outPaths;
}

void CreateTempFiles(const fs::path& dir, unsigned numFiles)
{
    auto files = GeneratePathNames(dir, numFiles);
    for (auto &oneFile : files)
    {
        std::ofstream entry(oneFile.replace_extension(".txt"));
        entry << "Hello World";
    }
}

void CreateTempData(const fs::path& tempPath, unsigned numDirectories, unsigned numFiles)
{
    fs::create_directory(tempPath);
    auto dirPaths = GeneratePathNames(tempPath, numDirectories);

    for (auto& dir : dirPaths)
    {
        if (fs::create_directory(dir))
            CreateTempFiles(dir, numFiles);
    }
}
```

In `CreateTempData()` we first create the root of our folder structure. Then we generate path names in `GeneratePathNames()`. Each pathname is built from a pointer address. Such an approach should give us good enough unique names. When we have a vector of unique paths, then we also generate another vector of paths for files. In this case, we use the `.txt` extension. While a directory is created using `fs::create_directory`, for files we can use standard stream objects to create them.

If we run the application, with the following parameters: `. temp 2 4` it will create the following directory structure:

```
temp
   tt22325368
      tt22283456.txt, size 11 bytes
      tt22283484.txt, size 11 bytes
```

```
      tt22283512.txt, size 11 bytes
      tt22283540.txt, size 11 bytes

   tt22325396
      tt22283456.txt, size 11 bytes
      tt22283484.txt, size 11 bytes
      tt22283512.txt, size 11 bytes
      tt22283540.txt, size 11 bytes
```

The complete code can be found below:

```
#include <chrono>
#include <filesystem>
#include <fstream>
#include <iostream>
#include <optional>
#include <sstream>
#include <string>
#include <vector>

namespace fs = std::filesystem;

std::vector<fs::path> GeneratePathNames(const fs::path& tempPath, unsigned num)
{
    std::vector<fs::path> outPaths{ num, tempPath };
    for (auto& dirName : outPaths)
    {
        const auto addr = reinterpret_cast<uintptr_t>(&dirName);
        dirName /= std::string("tt") + std::to_string(addr);
    }
    return outPaths;
}

void CreateTempFiles(const fs::path& dir, unsigned numFiles)
{
    auto files = GeneratePathNames(dir, numFiles);
    for (auto &oneFile : files)
    {
        std::ofstream entry(oneFile.replace_extension("txt"));
        entry << "Hello World";
    }
}

void CreateTempData(const fs::path& tempPath, unsigned numDirectories, unsigned numFiles)
{
    fs::create_directory(tempPath);
    auto dirPaths = GeneratePathNames(tempPath, numDirectories);

    for (auto& dir : dirPaths)
    {
        if (fs::create_directory(dir))
            CreateTempFiles(dir, numFiles);
    }
}

int main(int argc, char* argv[])
{
    try
```

```
    {
        const fs::path startingPath{ argc >= 2 ? argv[1] : fs::current_path() };
        const std::string strTempName{ argc >= 3 ? argv[2] : "temp" };
        const int numDirectories{ argc >= 4 ? std::stoi(argv[3]) : 4 };
        const int numFiles{ argc >= 5 ? std::stoi(argv[4]) : 4 };

        if (numDirectories < 0 || numFiles < 0)
            throw std::runtime_error("negative input numbers...");

        const fs::path tempPath = startingPath / strTempName;
        CreateTempData(tempPath, numDirectories, numFiles);
    }
    catch (const fs::filesystem_error& err)
    {
        std::cerr << "filesystem error! " << err.what() << '\n';
        if (!err.path1().empty())
            std::cerr << "path1: " << err.path1().string() << '\n';
        if (!err.path2().empty())
            std::cerr << "path2: " << err.path2().string() << '\n';
    }
    catch (const std::exception& ex)
    {
        std::cerr << "general exception: " << ex.what() << '\n';
    }
}
```

Filtering Files using Regex

The last example in this chapter will filter files with the addition of `std::regex` that is available since C++11.

The core of the `main()`:

```
const fs::path pathToShow{ argc >= 2 ? argv[1] : fs::current_path() };
const std::regex reg(argc >= 3 ? argv[2] : "");

auto files = CollectFiles(pathToShow);

std::sort(files.begin(), files.end());

for (auto& entry : files)
{
    const auto strFileName = entry.mPath.filename().string();
    if (std::regex_match(strFileName, reg))
        std::cout << strFileName << "\tsize: " << entry.mSize << '\n';
}
```

The application collects all the files from a given directory. Later a file entry is shown when it matches regex.

`CollectFiles()` function uses recursive directory iterator to find all the files and also build basic information about each file. In `main()` we sort each file by size.

```
struct FileEntry
{
    fs::path mPath;
    uintmax_t mSize{ 0 };

    static FileEntry Create(const fs::path& filePath) {
        return FileEntry{ filePath, fs::file_size(filePath) };
    }

    friend bool operator < (const FileEntry& a, const FileEntry& b) noexcept {
        return a.mSize < b.mSize;
    }
};

std::vector<FileEntry> CollectFiles(const fs::path& inPath)
{
    std::vector<fs::path> paths;
    if (fs::exists(inPath) && fs::is_directory(inPath))
    {
        std::filesystem::recursive_directory_iterator dirpos{ inPath };

        std::copy_if(begin(dirpos), end(dirpos), std::back_inserter(paths),
            [](const fs::directory_entry& entry) {
                return entry.is_regular_file();
            }
        );
    }
    std::vector<FileEntry> files(paths.size());
    std::transform(paths.cbegin(), paths.cend(), files.begin(), FileEntry::Create);
    return files;
}
```

In `CollectFiles` we use a recursive iterator and then `std::copy_if` to filter only regular files. Later, once the files are collected, we create the output vector of File Entries. `FileEntry::Create()` initialises objects with also the size of a file.

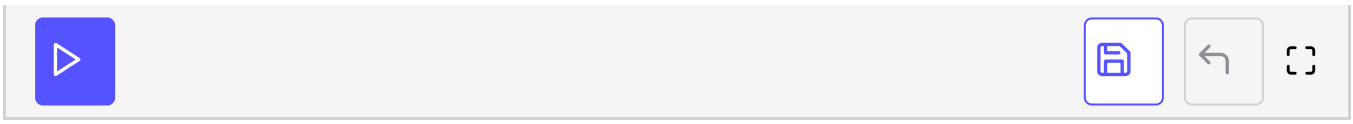
For example, if we run the application with the following parameters `temp .*.txt` we'll be looking for all `txt` files in a directory.

```
.\FilterFiles.exe temp .*.txt
tt22283456.txt  size: 11
tt22283484.txt  size: 11
tt22283512.txt  size: 11
tt22283540.txt  size: 11
tt22283456.txt  size: 11
tt22283484.txt  size: 11
tt22283512.txt  size: 11
tt22283540.txt  size: 11
```

Try it for yourself by running this code:

```
#include <algorithm>
#include <chrono>
#include <filesystem>
#include <iostream>
#include <regex>

namespace fs = std::filesystem;

struct FileEntry
{
    fs::path mPath;
    uintmax_t mSize{ 0 };

    static FileEntry Create(const fs::path& filePath) {
        return FileEntry{ filePath, fs::file_size(filePath) };
    }

    friend bool operator < (const FileEntry& a, const FileEntry& b) noexcept {
        return a.mSize < b.mSize;
    }
};

std::vector<FileEntry> CollectFiles(const fs::path& inPath)
{
    std::vector<fs::path> paths;
    if (fs::exists(inPath) && fs::is_directory(inPath))
    {
        std::filesystem::recursive_directory_iterator dirpos{ inPath };

        std::copy_if(begin(dirpos), end(dirpos), std::back_inserter(paths),
            [](const fs::directory_entry& entry) {
                return entry.is_regular_file();
            }
        );
    }
    std::vector<FileEntry> files(paths.size());
    std::transform(paths.cbegin(), paths.cend(), files.begin(), FileEntry::Create);
    return files;
}

int main(int argc, char* argv[])
{
    try
    {
        const fs::path pathToShow{ argc >= 2 ? argv[1] : fs::current_path() };
        const std::regex reg(argc >= 3 ? argv[2] : "");

        auto files = CollectFiles(pathToShow);

        std::sort(files.begin(), files.end());

        for (auto& entry : files)
        {
            const auto strFileName = entry.mPath.filename().string();
            if (std::regex_match(strFileName, reg))
                std::cout << strFileName << "\tsize: " << entry.mSize << '\n';
        }
    }
```

```
    catch (const fs::filesystem_error& err)
    {
        std::cerr << "filesystem error! " << err.what() << '\n';

        if (!err.path1().empty())
            std::cerr << "path1: " << err.path1().string() << '\n';
        if (!err.path2().empty())
            std::cerr << "path2: " << err.path2().string() << '\n';
    }
    catch (const std::exception& ex)
    {
        std::cerr << "general exception: " << ex.what() << '\n';
    }
}
```



These are all the things that you can get done using `std:filesystem` library!