Supporting Functions

Let's look at some additional functions the Filesytem library has to offer.

So far we covered three elements of the filesystem: the path class, `directory_entry` and `directory iterators`. The library also provides a set of non-member functions.

Query functions

function	description
`filesystem::is_block_file()`	checks whether the given path refers to block device
`filesystem::is_character_file()`	checks whether the given path refers to a character device
`filesystem::is_directory()`	checks whether the given path refers to a directory
	checks whether the given path

`filesystem::is_empty()`	refers to an empty file or directory
`filesystem::is_fifo()`	checks whether the given path refers to a named pipe
`filesystem::is_other()`	checks whether the argument refers to another file
`filesystem::is_regular_file()`	checks whether the argument refers to a regular file
`filesystem::is_socket()`	checks whether the argument refers to a named IPC socket
`filesystem::is_symlink()`	checks whether the argument refers to a symbolic link
`filesystem::status_known()`	checks whether file status is known
`filesystem::exists()`	checks whether path refers to existing file system object
`filesystem::file_size()`	returns the size of a file
`filesystem::last_write_time()`	gets or sets the time of the last data modification

Path related

function name	description
`filesystem::absolute()`	composes an absolute path
`filesystem::canonical(),`	composes a canonical path

`weakly_canonical()`	composes a canonical path
`filesystem::relativeproximate()`	composes a relative path
`filesystem::current_path()`	returns or sets the current working directory
`filesystem::equivalent()`	checks whether two paths refer to the same file system

Directory and files management

function name	description
`filesystem::copy()`	copies files or directories
`filesystem::copy_file()`	copies file contents
`filesystem::copy_symlink()`	copies a symbolic link
`filesystem::create_directory()`, `filesystem::create_directories()`	creates new directory
`filesystem::create_hard_link()`	creates a hard link
`filesystem::create_symlink()`, `filesystem::create_directory_symlink()`	creates a symbolic link
`filesystem::hard_link_count()`	returns the number of hard links referring to the specific file
`filesystem::permissions()`	modifies file access permissions

`filesystem::read_symlink()`	obtains the target of a symbolic link
`filesystem::remove()`, `filesystem::remove_all()`	removes a single file or whole directory recursively with all its content
`filesystem::rename()`	moves or renames a file or directory
`filesystem::resize_file()`	changes the size of a regular file by truncation or zero-fill
`filesystem::space()`	determines available free space on the file system
`filesystem::status()`, `filesystem::symlink_status()`	determines file attributes, determines file attributes, checking the symlink target
`filesystem::temp_directory_path()`	returns a directory suitable for temporary files

Getting & Displaying the File Time

In C++17 there's one thing about `last_write_time()` values that's inconvenient.

We have one free function and a method in `directory_entry`. They both return `file_time_type` which is currently defined as:

```
using file_time_type = std::chrono::time_point</*trivial-clock*/>;
```

From the standard, 30.10.25 Header `<filesystem>` synopsis:

> `trivial-clock` is an implementation-defined type that satisfies the TrivialClock requirements and that is capable of representing and

> measuring file time values. Implementations should ensure that the
>
> resolution and range of `file_time_type` reflect the operating system dependent resolution and range of file time values.

In other words, it’s implementation-dependent.

For example in GCC(up to 9.0)/Clang STL file time is implemented on top of `chrono::system_clock`, but in MSVC it’s some platform-specific clock.

Here are some more details about the implementation decisions in Visual Studio: std::filesystem::file_time_type does not allow easy conversion to time_t

The situation might soon improve as in C++20 we’ll get `std::chrono::file_clock` and also conversion routines between clocks. See P0355 (already added into C++20).

Let’s have a look at some code.

```
auto filetime = fs::last_write_time(myPath);
const auto toNow = fs::file_time_type::clock::now() - filetime;
const auto elapsedSec = duration_cast<seconds>(toNow).count();
// skipped std::chrono prefix for duration_cast and seconds
```

The above code gives you a way to compute the number of seconds that have elapsed since the last update. This is, however, not as useful as showing a real date.

On POSIX (GCC till 9.0 and Clang Implementation) you can easily convert file time to `system_clock` and then obtain `std::time_t`:

```
auto filetime = fs::last_write_time(myPath);
std::time_t convfiletime = std::chrono::system_clock::to_time_t(filetime);
std::cout << "Updated: " << std::ctime(&convfiletime) << '\n';
```

In MSVC the code won’t compile. However there’s a guarantee that `file_time_type` is usable with native OS functions that takes `FILETIME`. So we can write the following code to solve the issue:

```
auto filetime = fs::last_write_time(myPath);
FILETIME ft;
```

```
memcpy(&ft, &filetime, sizeof(FILETIME));
SYSTEMTIME  stSystemTime;
if (FileTimeToSystemTime(&ft, &stSystemTime)) {

    // use stSystemTime.wYear, stSystemTime.wMonth, stSystemTime.wDay, ...
}
```

In GCC 9.1 we can use the following trick:

```
auto convFileTime = std::chrono::time_point_cast<std::chrono::system_clock::duration>(fileTime - fs::file_time_type::clock::now() + std::chrono::system_clock::now());
std::time_t convfiletime = std::chrono::system_clock::to_time_t(convFileTime);
return std::ctime(&convfiletime);
```

Examples

The code below covers a few examples of how we can use the supporting functions that we have mentioned above. The directory of this code is called **usercode** and the file which is running the code below is **input.cpp**.

input.cpp

test.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);
        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";
    }
}
```

Now let's learn about functions related to file permissions.