Error Handling & File Races

Let's briefly discuss how errors are handled in std:filesystem library.

WE'LL COVER THE FOLLOWING

- File Races

So far the examples in this chapter used exception handling as a way to report errors. The filesystem API is also equipped with function and method overloads that outputs an error code.

You can decide if you want exceptions or error codes.

For example, we have two overloads for `file_size`:

```
uintmax_t file_size(const path& p);
uintmax_t file_size(const path& p, error_code& ec) noexcept;
```

the second one can be used in the following way:

```
const std::filesystem::path testPath("C:\test.txt");
std::error_code ec{};
auto size = std::filesystem::file_size(testPath, ec);
if (ec == std::error_code{})
    std::cout << "size: " << size << '\n';
else
    std::cout << "error when accessing test file, size is: "
              << size << " message: " << ec.message() << '\n';
```

`file_size` takes additional output parameter - `error_code` and will set a value if something happens. If the operation is successful then `ec` will be value initialized.

File Races

It’s important to point out the undefined behaviour that might happen when a file race occurs.

From *30.10.2.3 File system race behaviour*:

> The behavior is undefined if the calls to functions in this library introduce a file system race.

And from *30.10.9 file system race*:

> The condition that occurs when multiple threads, processes, or computers interleave access and modification of the same object within a file system.

Now let’s look at a few use cases where `std:filesystem` can be used!