The Path Object

Let's explore the Path Object in detail!

WE'LL COVER THE FOLLOWING

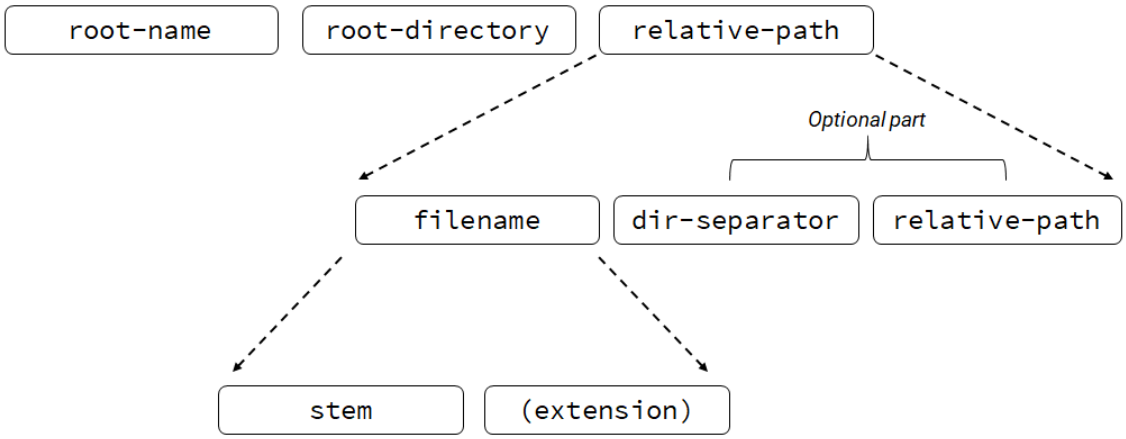
The core part of the library is the `path` object. It contains a pathname - a string that forms the name of the path. The object doesn't have to point to an existing file in the filesystem. The path might be even in an invalid form.

The path is composed of the following elements:

`root-name root-directory relative-path`:

- (optional) root-name: POSIX systems don't have a root name. On Windows, it's usually the name of a drive, like `"C:"`
- (optional) root-directory: distinguishes relative path from the absolute path
- relative-path:
  - filename
  - directory separator
  - relative-path

We can illustrate it with the following diagram:



The class implements a lot of methods that extracts the parts of the path:

Method	Description
`path::root_name()`	returns the root-name of the path
`path::root_directory()`	returns the root directory of the path
`path::root_path()`	returns the root path of the path
`path::relative_path()`	returns path relative to the root path
`path::parent_path()`	returns the path of the parent path
`path::filename()`	returns the filename path component
`path::stem()`	returns the stem path component
`path::extension()`	returns the file extension path component

If a given element is not present, then the above functions return an empty

path.

There are also methods that query elements of the path:

Query name	Description
`path::has_root_path()`	queries if a path has a root
`path::has_root_name()`	queries if a path has a root name
`path::has_root_directory()`	checks if a path has a root directory
`path::has_relative_path()`	checks if a path has a relative path component
`path::has_parent_path()`	checks if a path has a parent path
`path::has_filename()`	checks if a path has filename
`path::has_stem()`	checks if a path has a stem component
`path::has_extension()`	checks if a path has an extension

We can use all of the above methods and compose an example that shows info about a given path:

```
const filesystem::path testPath{...};

if (testPath.has_root_name())
    cout << "root_name() = " << testPath.root_name() << '\n';
else
    cout << "no root-name\n";

if (testPath.has_root_directory())
    cout << "root directory() = " << testPath.root_directory() << '\n';
else
    cout << "no root-directory\n";

if (testPath.has_root_path())
```

```
if (testPath.has_root_path())
    cout << "root_path() = " << testPath.root_path() << '\n';
else
    cout << "no root-path\n";

if (testPath.has_relative_path())
    cout << "relative_path() = " << testPath.relative_path() << '\n';
else
    cout << "no relative-path\n";

if (testPath.has_parent_path())
    cout << "parent_path() = " << testPath.parent_path() << '\n';
else
    cout << "no parent-path\n";

if (testPath.has_filename())
    cout << "filename() = " << testPath.filename() << '\n';
else
    cout << "no filename\n";

if (testPath.has_stem())
    cout << "stem() = " << testPath.stem() << '\n';
else
    cout << "no stem\n";

if (testPath.has_extension())
    cout << "extension() = " << testPath.extension() << '\n';
else
    cout << "no extension\n";
```

If you run this code on your system, here's what you will see as output for a file path like `"C:\Windows\system.ini"`:

```
root_name() = "C:"
root directory() = "\\"
root_path() = "C:\\"
relative_path() = "Windows\\system.ini"
parent_path() = "C:\\Windows"
filename() = "system.ini"
stem() = "system"
extension() = ".ini"
```

Similarly on POSIX systems, for a path `/usr/temp/abc.txt`:

```
no root-name
root directory() = "/"
root_path() = "/"
relative_path() = "usr/temp/abc.txt"
parent_path() = "/usr/temp"
filename() = "abc.txt"
stem() = "abc"
extension() = ".txt"
```

There's also a trick that allows you to iterate over the parts of a path object. `std::filesystem::path` implements overloads for `begin()` and `end()` and that's why you can use it in a range based for loop:

```
int i = 0;
for (const auto& part : testPath)
    cout << "path part: " << i++ << " = " << part << '\n';
```

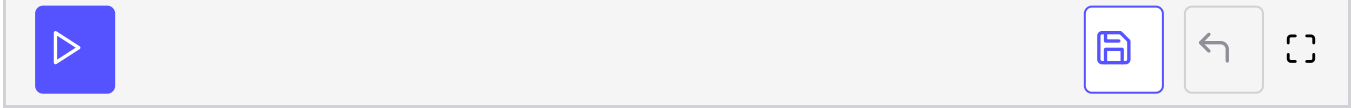
In this case, the output for path `C:\Windows\system.ini` will be:

```
path part: 0 = C:
path part: 1 = \
path part: 2 = Windows
path part: 3 = system.ini
```

Here's what the final code will look like after compiling all the snippets from above:

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

namespace fs = std::filesystem;

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

        std::cout << "path::preferred_separator: ";
        if constexpr (std::is_same_v<fs::path::value_type, wchar_t>)
            std::wcout << "wchar_t: " << fs::path::preferred_separator << L'\n';
        else
            std::cout << fs::path::preferred_separator << '\n';

        std::cout << "exists() = " << fs::exists(testPath) << '\n';

        if (testPath.has_root_name())
            std::cout << "root_name() = " << testPath.root_name() << '\n';
        else
            std::cout << "no root-name\n";

        if (testPath.has_root_directory())
            std::cout << "root directory() = " << testPath.root_directory() << '\n';
        else
            std::cout << "no root-directory\n";
```

```
        if (testPath.has_root_path())
            std::cout << "root_path() = " << testPath.root_path() << '\n';

        else
            std::cout << "no root-path\n";

        if (testPath.has_relative_path())
            std::cout << "relative_path() = " << testPath.relative_path() << '\n';
        else
            std::cout << "no relative-path\n";

        if (testPath.has_parent_path())
            std::cout << "parent_path() = " << testPath.parent_path() << '\n';
        else
            std::cout << "no parent-path\n";

        if (testPath.has_filename())
            std::cout << "filename() = " << testPath.filename() << '\n';
        else
            std::cout << "no filename\n";

        if (testPath.has_stem())
            std::cout << "stem() = " << testPath.stem() << '\n';
        else
            std::cout << "no stem\n";

        if (testPath.has_extension())
            std::cout << "extension() = " << testPath.extension() << '\n';
        else
            std::cout << "no extension\n";

        int i = 0;
        for (const auto& part : testPath)
            std::cout << "path part: " << i++ << " = " << part << '\n';

        auto pathAbsolute = fs::absolute(testPath);
        std::cout << "absolute: " << pathAbsolute.string() << '\n';

        pathAbsolute.make_preferred();
        std::cout << "preferred: " << pathAbsolute.string() << '\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';
        }
}
```



Path Operations

Below you can find a table with other important methods of the `path` class:

Operation	Description
`path::append()`	appends one path to the other, with a directory separator
`path::concat()`	concatenates the paths, without a directory separator
`path::clear()`	erases the elements and makes it empty
`path::remove_filename()`	removes the filename part from a path
`path::replace_filename()`	replaces a single filename component
`path::replace_extension()`	replaces the extension
`path::swap()`	swaps two paths
`path::compare()`	compares the lexical representations of the path and another path, returns an integer
`path::empty()`	checks if the path is empty

Comparison

The `path` class has several overloaded operators:

`==`, `!=`, `<`, `>`, `<=`, `=>`

And the `path::compare()` method, which returns an integer value.

All methods compare element by element, using the native format of the path

All methods compare element by element, using the native format of the path.

```
fs::path p1 { "/usr/a/b/c" };
fs::path p2 { "/usr/a/b/c" };
assert(p1 == p2);
assert(p1.compare(p2) == 0);

p1 = "/usr/a/b/c";
p2 = "/usr/a/b/c/d";
assert(p1 < p2);
assert(p1.compare(p2) < 0);
```

And on Windows we can also test the cases where we have a root element in a path:

```
p1 = "C:/test";
p2 = "abc/xyz"; // no root path, so it's "less" than a path with a root
assert(p1 > p2);
assert(p1.compare(p2) > 0);
```

Or, also on Windows, a case where path formats are different:

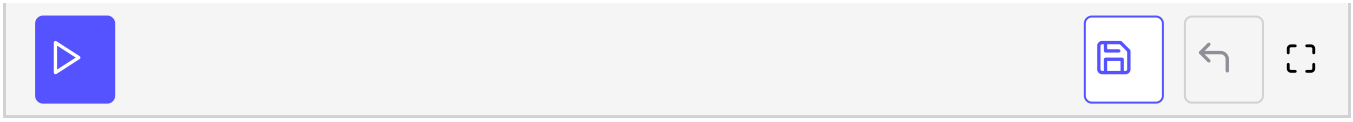
```
fs::path p3 { "/usr/a/b/c" }; // on Windows it's converted to native format
fs::path p4 { "\\usr/a\\b/c" };
assert(p3 == p4);
assert(p3.compare(p4) == 0);
```

Playground

Here's the code for your reference:

```
#include <cassert>
#include <filesystem>

namespace fs = std::filesystem;

int main()
{
  #ifdef _MSC_VER
  fs::path p1 = "C:/test";
  fs::path p2 = "C:/abc";
  assert(p1 > p2);
  assert(p1.compare(p2) > 0);

  p1 = "C:/test";
  p2 = "abc/xyz";
  assert(p1.has_root_directory());
  assert(p2.is_relative());
  assert(p1 > p2);
  assert(p1.compare(p2) > 0);
```

```
  fs::path p3 = "/usr/a/b/c"; // on Windows it's converted to native format
  fs::path p4 = "\\usr/a\\b/c";

  assert(p3 == p4);
  assert(p3.compare(p4) == 0);
  #endif

  fs::path p5 = "/usr/a/b/c";
  fs::path p6 = "/usr/a/b/c";
  assert(p5 == p6);
  assert(p5.compare(p6) == 0);

  p5 = "/usr/a/b/c";
  p6 = "/usr/a/b/c/d";
  assert(p5 < p6);
  assert(p5.compare(p6) < 0);
}
```



In the next lesson, we will learn how to compose a path in the form of a String `C++`.