Path Composition

C++ allows us to create a path in the form of a string. Let's find out how.

WE'LL COVER THE FOLLOWING

We have two methods that allow to compose a path:

- `path::append()` - adds a path with a directory separator.
- `path::concat()` - only adds the 'string' without any separator.

The functionality is also available with operators `/`, `/=` (append), `+` and `+=` (concat).

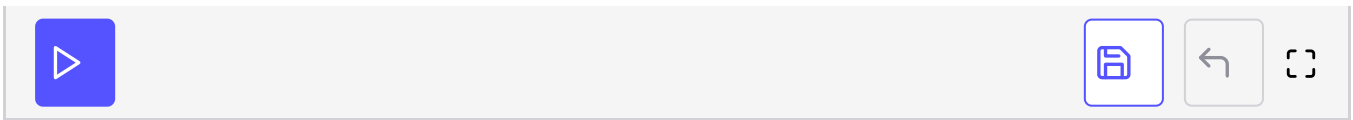
For example:

```
#include <iostream>
#include <filesystem>
using namespace std;

int main(){

  namespace fs = std::filesystem;

  fs::path p1("C:\\temp"); p1 /= "user";
  p1 /= "data";
  cout << p1 << '\n';
  fs::path p2("C:\\temp\\"); p2 += "user";
  p2 += "data";
  cout << p2 << '\n';
}
```



However, appending a path has several rules that you have to be aware of.

For example, if the other path is absolute or the other path has a root-name, and the root-name is different from the current path root name. Then the append operation will replace `this`.

```
auto resW = fs::path{"foo"} / "D:\"; // Windows
auto resP = fs::path{"foo"} / "/bar"; // POSIX
// resW is "D:\" now
// resP is now "/bar"
```

In the above case `resW` and `resP` will contain the value from the second operand. As `D:\` and `/bar` contains root elements.

What can we do more? Let's find a file size (using `file_size`):

```
#include <iostream>
#include <filesystem>
using namespace std;

namespace fs = std::filesystem;

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

int main(){
  fs::path pathToCheck("./test.txt");
  cout << "File Size: " << ComputeFileSize(pathToCheck);
}
```

As additional information, most of the functions that work on a path have two versions:

- One that throws: `filesystem_error`
- Another with `error_code` (system specific)

Stream Operators

The `path` class also implements `>>` and `<<` operators.

The operators use `std::quoted` to preserve the correct format. That's why the paths in the examples showed quotes.

On Windows, this will also cause to output `"\\"` for paths in native format.

For example on POSIX:

```
fs::path p1 { "/usr/test/temp.xyz" };
std::cout << p1;
```

The code will print `"/usr/test/temp.xyz"`.

And on Windows:

```
fs::path p1{ "usr/test/temp.xyz" };
fs::path p2{ "usr\\test\\temp.xyz" };
std::cout << p1 << '\n' << p2;
```

The code will output:

```
"usr/test/temp.xyz"
"usr\\test\\temp.xyz"
```

Path formats and conversion

The filesystem library is modelled on top of POSIX (for example all Unix Based systems implements POSIX standard), but also works with other filesystems, for instance with Windows. Because of that, there are few things to keep in mind when using paths in a portable way.

The first thing is the path format. We have two core modes:

- generic - generic format, the format as specified by the standard (based on the POSIX format)
- native - format used by the particular implementation

In POSIX systems native format is equal to generic. But On Windows it's different.

The main difference of the format is that Windows uses backslashes (`\`)

rather than slashes (`/`). Another point is that Windows has a root directory - like `C:`, `D:` or other drive letters.

The other important aspect is the string type that is used to hold path elements. In POSIX it's `char` (and `std::string`), but on Windows it's `wchar_t` and `std::wstring`. The `path` type specifies `path::value_type` and `string_type` (defined as `std::basic_string<value_type>`) to expose those properties.

The `path` class has several methods that allow you to use the best matching format.

If you want to work with the native format you can use:

Operation	Description
`path::c_str()`	returns `value_type*`
`path::native()`	returns `string_type&`

And there are many methods that will convert the native format:

Operation	Description
`path::string()`	converts to `string`
`path::wstring()`	converts to `wstring`
`path::u8string()`	converts to `u8string`
`path::u16string()`	converts to `u16string`
`path::u32string()`	converts to `u32string`

> Since Windows uses `wchar_t` as the underlying type for paths, then you need to be aware of "hidden" conversions to `char`.

That’s all for path composition, now let’s learn how to traverse over a path in C++.