Filesystem library

This newly added feature lets us interact with the directories and files on our system.

The new filesystem library is based on boost::filesystem. Some of its components are optional. This means not all functionality of std::filesystem is available on each implementation of the filesystem library. For example, FAT-32 does not support symbolic links.

> **i Using cppreference.com**
> At the time of the writing this book (October 2017), I had no C++±compiler at my disposal that supports the new filesystem library; therefore, I executed the programs in this chapter with the newest GCC-compiler on cppreference.com: filesystem that supports the new filesystem library.
>
> To use the new filesystem library you may have to include and apply the experimental namespace.
>
> ```
> #include <experimental/filesystem>
> namespace fs = std::experimental::filesystem;
> ```
>
> All examples in this chapter are written without the experimental namespace.

The library is based on the three concepts: file, file name, and path.

- A *file* is an object that holds data such that you can write to it or read from it. A file has a name and a file type. A file type can be a directory, hard link, symbolic link or a regular file.
    - A *directory* is a container for holding other files. The current directory is represented by a dot `"."`; the parent directory is represented by two dots `".."`.

    - A *hard link* associates a name with an existing file.
    - A *symbolic link* associates a name with a path that may exist.
    - A *regular file* is a directory entry which is neither a directory, a hard link, nor a symbolic link.
- A *file name* is a string that represents a file. It is implementation-defined which characters are allowed, how long the name could be or if the name is case sensitive.
- A *path* is a sequence of entries that identify the location for a file. It has an optional root-name such a “C:” on Windows, followed by a root-directory such a “/” on Unix. Additional parts can be directories, hard links, symbolic links, or regular files. A path can be absolute, canonical, or relative.
    - An *absolute path* is a path that identifies a file.
    - A *canonical path* is a path that includes neither a symbolic link nor the relative paths `"."` (current directory) or `".."` (parent directory).
    - A *relative path* specifies a path relative to a location in the file system. Paths such as `"."` (current directory), `".."` (parent directory) or `"home/rainer"` are relative paths. On Unix, they do not start at the root-directory `"/"`.

Here is an introductory example to the filesystem.

```
#include <fstream>
#include <iostream>
#include <string>
#include <filesystem>
namespace fs = std::filesystem;

int main(){

    std::cout << "Current path: " << fs::current_path() << std::endl;

    std::string dir= "sandbox/a/b";
    fs::create_directories(dir);

    std::ofstream("sandbox/file1.txt");
    fs::path symPath= fs::current_path() /=  "sandbox";
    symPath /= "syma";
    fs::create_symlink("a", "symPath");
```

```
    std::cout << "fs::is_directory(dir): " << fs::is_directory(dir) << std::endl;
    std::cout << "fs::exists(symPath): "  << fs::exists(symPath) << std::endl;
    std::cout << "fs::symlink(symPath): " << fs::is_symlink(symPath) << std::endl;



    for(auto& p: fs::recursive_directory_iterator("sandbox"))
        std::cout << p << std::endl;
    fs::remove_all("sandbox");

}
```

Overview to the filesystem library

`fs::current_path()` (1) returns the current path. You can create a directory hierarchy (2) with `std::filesystem::create_directories`. The `/=` operator is overloaded for a path (3). Therefore, I can directly create a symbolic link (4) or check the properties of a file. The call `recursive_directory_iterator` (5) allows you to traverse directories recursively.

Output:

```
Current path: "/tmp/1469540273.75652"

fs::is_directory(dir): true
fs::exists(symPath): true
fs::symlink(symPath): true

"sandbox/syma"
"sandbox/file1.txt"
"sandbox/a"
"sandbox/a/b"
"sandbox/a/b/c"
```