Chapter 13 - The Filesystem

The os module is one of the bread and butter workhorses of python. We are going to cherry pick some functions out of the os module and figure out what their analogs are in c++. Lets get started.

Basic Os Goodness

Our cup runneth over. Turns out, there are about 220 symbols exposed in the top os python module alone. I doubt that you or I have the patience to explore each one. So we are going to explore boost based on themes addressed in the os python module, rather than looking for an analog to each function.

environment variable access

Well this one is simple. the `cstdlib` header provides a function called `getenv()` which takes a c string with the name of the variable and returns a c-string with the result, or NULL if the call isn't valid.

```
#include <iostream>
#include <stdlib.h>     /* getenv */

using std::cout; using std::endl;
int main ()
{
  char* pPath;
  pPath = getenv ("PATH");
  if (pPath!=nullptr)
    cout << "env var: " << pPath << endl;
  return 0;
}
```

file and path manipulation

Unlike Python, there currently isn't a built in way of manipulating files and paths in a cross platform manner. We do have the next best thing: Boost::filesystem. Or rather we can download and build it. You might want to use conan for this as it should be far simpler and less time consuming than buidling Boost libraries....

Boost - what is it?

First, what is Boost? Boost is one of the oldest and most well regarded libraries for C++ in existance. Many of the sub projects in Boost are finding their way

into the standard library. In fact, Boost::filesystem, at which we are about to look, will essentially be absorbed into C++17.

getting the size of a file

```
#include <iostream>
#include <boost/filesystem.hpp>

int main(int argc, char* argv[])
{
  if (argc < 2)
  {
    std::cout << "Usage: tut1 path\n";
    return 1;
  }

  std::cout << argv[1] << " " << boost::filesystem::file_size(argv[1]) << '\n';
  return 0;
}
```

status queries

Boost includes calls to determine if a resource exists, is a file, a directory,etc...

Most tests operate on boost path objects, so first we create one. In this case we construct path with argv[1], but we could be using a std::string or a c stringl

```
// create a path object, assuming that the user has specified a path as an input
boost::filesystem::path p(argv[1]);
```

Test to see if path exists:

```
std::cout << argv[1] << " exists? " << boost::filesystem::exists(p) << std::endl;;
```

Test to see if path is a directory

```
std::cout << argv[1] << " is a directory? " << boost::filesystem::is_directory(p) << std::en
```

Test to see if path is a regular file:

```
std::cout << argv[1] << "is regular file? " << boost::filesystem::is_regular_file(p) << std
```

Test to see if a path is a symlink

```
stdd::cout << argv[1] << "is symlink? " << boost::filesystem::is_symlink(p) << std::endl;
```

Iterating over directory contents

Boost provides a *directory_iterator* class to handle this for us.

```
if( exists( directory ) )
   {
      directory_iterator end ;

      for( directory_iterator iter(directory) ; iter != end ; ++iter ) {
        if ( !is_directory( *iter ) )
        {
            // this is a file
            cout << iter->path;

            // rest of the code
        }
      }
   }
```

decomposing a path

Much like Python's os module, Boost's path class has a wealth of decomposition methods. All the following examples assume a path object instantiated with a filepath:

```
boost::filesystem::path p(pathstr);
```

root name

For those of you dealing with windows paths, my condolences. The root_name method will return the drive letter given a windows path. So given path p("c:\foo\bar"), p.root_name() returns "c:".

```
cout << p.root_name() << endl;
```

root directory

Continuing on with our windows path blues, the root_directory returns the base directory under the drive letter. So, given "c:\foo\bar", p.root_directory() returns "";

```
cout << p.root_directory() << endl;
```

root path

Rounding out our windows trifecta, root_path returns the root name and root directory.

```
cout << p.relative_path() << endl;
```

dirname

Like os.path.dirname in python, the path class has a parent_path() method which returns the parent directory. Path also has a `has_parent_path()` method...

```
if(p.has_parent_path())
    cout << p.parent_path() << endl;
```

basename

Again like os.path.basename in python, the path class has a `filename()` method to retrieve the file name from a full path. It also has a `has_filename()` method.

```
if(p.has_filename())
    cout << p.filename() << endl;
```

file stem

There is no direct analog in python. The path class will retrieve the base filename if available when calling `stem()`. And, as with other calls, there is a corresponding `has_stem()` method.

```
if(p.has_stem())
    cout << p.stem() << endl; // python's basename
```

extension

Lastly, like os.path.extension, the path class has an `extension()` method which returns the file extension for a path. And, like the other methods, it also has a `has_extension()` method to test for existence.

```
if(p.has_extension())
    cout << p.extension() << endl;;
```

Iterating over path components

You might be wondering why we have to use a directory iterator to iterate over directory contents. Especially given our past experience with container iterators. The answer is that you can iterate over a path. However, this returns a path's components, not its contents:

```
boost::filesystem::path p("/dd/dept/software/users/jgerber/foobar.txt");
for(auto &parts::p)
    std::cout << p << " ";
cout << endl;
```

composing a path

Boost has a nice overload for building up a path. you can use `/=` to append a directory or file to an existing path...

```
boost::filesystem::path p;
for(auto i=0; i < argc; i++)
    p /= argv[i];
```

additional path methods

is_absollute

How do you tell if a path is absolute or relative? `is_absolute()`

empty path?

How about testing if a path is empty? `empty()`

normpath analog

If you are familiar with os.path.normpath, you are probably wondering where this functionality lies in boost. That is, if you want to get rid of dots, extra slashes, etc in a path, you can use the `boost::filesystem::canonical()` method, provided that the path actually exists. That last bit is important. While os.path.normpath in python will operate on a string regardless of whether or not it represents a valid path, canonical will only operate on paths which exist on disk. Otherwise it will throw.

```
boost::filesystem::canoncial("/usr/bin/../bin");
```

Additional Free Functions

A note on Boost free function error reporting. Boost functions generally have two variants - one which throws an exception, and one which takes a reference to an error code. (system::error_code). The latter populates the error code when problems are encountered in lieu of throwing an exception. Which you prefer is a matter of style I suppose. I am only going to show the exception throwing variety. Not out of preference, but out of laziness.

hard link count

stat has the number of links for a given file. Boost has `hard_link_count(const path&)`

last modification time

```
std::time_t last_write_time(const path& p);
```

getting permissions

```
void permissions(const path& p, perms prms)
```

delete file

```
bool remove(const path& p);
```

delete directory and contents

The function returns the number of files removed...
```
uintmax_t remove_all(const path& p);
```

rename a file or directory

```
void rename(const path& old_p, const path& new_p);
```

create a temp directory

```
path temp_directory_path();
```

create a temp directory with control over naming

Each occurrence of a percent sign character is replaced by a random hexadecimal digit character in the range 0-9, a-f.

```
path unique_path(const path& model="%%%%-%%%%-%%%%-%%%%");
```

create a directory

```
bool create_directory(const path& p);
```

returns success or failure.

create multiple directories in a path.

like mkdir -p

```
bool create_directories(const path& p);
```

create a directory symlink

```
void create_directory_symlink(const path& to, const path& new_symlink);
```

create a file symlink

```
void create_symlink(const path& to, const path& new_symlink);
```

create a hardlink

```
void create_hard_link(const path& to, const path& new_hard_link);
```

Homework

1. In the section on environment variables, we presented a function in the standard library to retrieve environemnt variable contents. Unfortunately, the return of said function is of the following form:

   FOO=BAR

We are usually only interested in the results, not the key. Write a function which accepts the name of a variable as its argument and returns a string as its result. Return only the value (ie the BAR poriton from above). If the variable does

not exist, return “”. (You probably don’t readily know exactly what to do. Here is a hint: Google splitting std::string or c string)