# The File System

**Working with files and directories.**

# Overview

- Working with paths
- Basic file operations
- Working with directories
- Finding files with glob patterns
- Temporary files

# Paths

- Different operating systems use different notations to specify the location of a file or directory (in other words, a path).
- This makes it hard to work with files and directories in a portable way.
- POCO provides a class that abstracts the differences in the various notations, by focusing on common concepts.
- POCO supports the path notations used by:
  - Windows
  - Unix
  - OpenVMS

# Paths in POCO

- A path in POCO consists of:
  - an optional node name:
    on Windows, this is the computer name in an UNC path;
    on OpenVMS, this is the node name of a system in a cluster;
    on Unix, this is unused.
  - an optional device name:
    on Windows, this is a drive letter;
    on OpenVMS, this is the name of a storage device;
    on Unix, this is unused.
  - a list of directory names
  - a file name (including extension) and version (OpenVMS only)

# Paths in POCO (cont'd)

- POCO supports two kinds of paths:
  - absolute path
    describes the path to a resource, starting at a filesystem root.
  - relative path
    describes the path to a resource, starting at a certain directory (usually, the user's current directory).
    A relative path can be transformed into an absolute one (but not vice versa).
- Also, a path can point to a directory, or to a file.
  For a file pointing to a directory, the file name part is empty.

# Path Examples

| | |
|---|---|
| Path: | C:\Windows\system32\cmd.exe |
| Style: | Windows |
| Kind: | absolute, to file |
| Node Name: | – |
| Device Name: | C |
| Directory List: | Windows, system32 |
| File Name: | cmd.exe |
| File Version: | – |

# Path Examples (cont'd)

| | |
|---|---|
| Path: | Poco\Foundation\ |
| Style: | Windows |
| Kind: | relative, to directory |
| Node Name: | – |
| Device Name: | – |
| Directory List: | Poco, Foundation |
| File Name: | – |
| File Version: | – |

# Path Examples (cont'd)

| | |
|---|---|
| Path: | \\www\site\index.html |
| Style: | Windows |
| Kind: | absolute, to file |
| Node Name: | www |
| Device Name: | – |
| Directory List: | site |
| File Name: | index.html |
| File Version: | – |

# Path Examples (cont'd)

| | |
|---|---|
| Path: | /usr/local/include/Poco/Foundation.h |
| Style: | Unix |
| Kind: | absolute, to file |
| Node Name: | – |
| Device Name: | – |
| Directory List: | usr, local, include, Poco |
| File Name: | index.html |
| File Version: | – |

# Path Examples (cont'd)

| | |
|---|---|
| Path: | ../bin/ |
| Style: | Unix |
| Kind: | relative, to directory |
| Node Name: | – |
| Device Name: | – |
| Directory List: | .., bin |
| File Name: | – |
| File Version: | – |

# Path Examples (cont'd)

| | |
|---|---|
| Path: | VMS001::DSK001:[POCO.INCLUDE.POCO]POCO.H;2 |
| Style: | OpenVMS |
| Kind: | absolute, to file |
| Node Name: | VMS001 |
| Device Name: | DSK001 |
| Directory List: | POCO, INCLUDE, POCO |
| File Name: | POCO.H |
| File Version: | 2 |

# The Path Class

- Poco::Path represents a path in POCO.
- #include "Poco/Path.h"
- Poco::Path does not care whether the path it contains actually exists in the file system (this is what Poco::File is for, which will be discussed shortly).
- Poco::Path supports value semantics (copy construct and assignment); relational operators are not available.

# Building a Path

- There are two ways to build a path:
  - build one from scratch, piece by piece
  - build one by parsing a string containing a path; you have to specify the style of the path:
    - PATH_UNIX
    - PATH_WINDOWS
    - PATH_VMS
    - PATH_NATIVE (the native syntax of the current system)
    - PATH_GUESS (let POCO figure out the syntax)

# Building a Path From Scratch

1. Create an empty path, using the default constructor (relative path), or the constructor taking a boolean argument (true = absolute, false = relative).

2. Use the following mutators to set node and device, if required:
   void setNode(const std::string& node)
   void setDevice(const std::string& device)

3. Add directory names:
   void pushDirectory(const std::string& name)

4. Set the file name:
   void setFileName(const std::string& name)

```
#include "Poco/Path.h"

int main(int argc, char** argv)
{
	Poco::Path p(true); // path will be absolute
	p.setNode("VMS001");
	p.setDevice("DSK001");
	p.pushDirectory("POCO");
	p.pushDirectory("INCLUDE");
	p.pushDirectory("POCO");
	p.setFileName("POCO.H");

	std::string s(p.toString(Poco::Path::PATH_VMS));
		// "VMS001::DSK001:[POCO.INCLUDE.POCO]POCO.H"

	p.clear(); // start over with a clean state
	p.pushDirectory("projects");
	p.pushDirectory("poco");

	s = p.toString(Poco::Path::PATH_WINDOWS); // "projects\poco\"
	s = p.toString(Poco::Path::PATH_UNIX);    // "projects/poco/"
	s = p.toString(); // depends on your platform

	return 0;
}
```

# Parsing a Path From a String

- A path can be constructed from a string (std::string or old-style C string) containing a path in any supported format:
  Path(const std::string& path)
  Path(const std::string& path, Style style)
  if no style is given, the path is expected to be in native format.

- A path can also be constructed from another path (pointing to a directory) and a file name, or from two paths (the first one absolute, the second one relative):
  Path(const Path& parent, const std::string& fileName)
  Path(const Path& parent, const Path& relative)

# Parsing a Path From a String (cont'd)

- A path can also be created by assigning a string, using the assignment operator, or the assign() or parse() methods:
  Path& assign(const std::string& path)
  Path& parse(const std::string& path)
  Path& assign(const std::string& path, Style style)
  Path& parse(const std::string& path, Style style)

- If a path is not valid, a Poco::PathSyntaxException is thrown. To test whether a path's syntax is valid, tryParse() can be used:
  bool tryParse(const std::string& path)
  bool tryParse(const std::string& path, Style style)

```
#include "Poco/Path.h"

using Poco::Path;

int main(int argc, char** argv)
{
    //creating a path will work independent of the OS
    Path p("C:\\Windows\\system32\\cmd.exe");
    Path p("/bin/sh");

    p = "projects\\poco";
    p = "projects/poco";

    p.parse("/usr/include/stdio.h", Path::PATH_UNIX);

    bool ok = p.tryParse("/usr/*/stdio.h");
    ok = p.tryParse("/usr/include/stdio.h", Path::PATH_UNIX);
    ok = p.tryParse("/usr/include/stdio.h", Path::PATH_WINDOWS);
    ok = p.tryParse("DSK$PROJ:[POCO]BUILD.COM", Path::PATH_GUESS);


    return 0;
}
```

# Working With Paths

- A Poco::Path can be converted to a string:
  std::string toString()
  std::string toString(Style style)
- It is possible to access the different parts of a path:
  const std::string& getNode()
  const std::string& getDevice()
  const std::string& directory(int n) (also operator [])
  const std::string& getFileName()
- To get the number of directories in a path:
  int depth() const

# Working With Paths (cont'd)

- The two parts of a file name, base name and extension, can be accessed, too:
  - std::string getBaseName() const
    void setBaseName(const std::string& baseName)
  - std::string getExtension() const
    void setExtension(const std::string& extension)
- The base name ends before (and the extension starts after) the last period in the file name.

```
#include "Poco/Path.h"

using Poco::Path;

int main(int argc, char** argv)
{
	Path p("c:\\projects\\poco\\build_vs80.cmd", Path::PATH_WINDOWS);

	std::string device(p.getDevice());		// "c"
	int n = p.depth();				// 2
	std::string dir1(p.directory(0));		// "projects"
	std::string dir2(p[1]);				// "poco"
	std::string fileName(p[2]);			// "build_vs80.cmd"
	fileName = p.getFileName();
	std::string baseName(p.getBaseName());	// "build_vs80"
	std::string extension(p.getExtension()); // "cmd"

	p.setBaseName("build_vs71");
	fileName = p.getFileName();			// "build_vs71.cmd"

	return 0;
}
```

# Path Operations

- Path& makeDirectory()
  makes the file name (if there is one) the last directory name
- Path& makeFile()
  makes the last directory name the file name, if there is none
- Path& makeParent()
  Path parent() const
  makes the path refer to its parent (clears the file name, if there is one, otherwise removes the last directory)

# Path Operations (cont'd)

- Path& makeAbsolute()
  Path& makeAbsolute(const Path& base)
  Path absolute() const
  Path absolute(const Path& base)
  turns a relative path into an absolute one
- Path& append(const Path& path)
  appends another path
- Path& resolve(const Path& path)
  if path is absolute, it replaces the current one; otherwise it is appended

# Path Properties

- bool isAbsolute() const
  returns true if the path is absolute, false otherwise
- bool isRelative() const
  return true if the path is relative, false otherwise
- bool isDirectory() const
  returns true if the path does not have a file name
- bool isFile() const
  returns true if the path has a file name

```
#include "Poco/Path.h"

using Poco::Path;

int main(int argc, char** argv)
{
	Path p("/usr/include/stdio.h", Path::PATH_UNIX);

	Path parent(p.parent());
	std::string s(parent.toString(Path::PATH_UNIX)); // "/usr/include/"

	Path p1("stdlib.h");
	Path p2("/opt/Poco/include/Poco.h", Path::PATH_UNIX);

	p.resolve(p1);
	s = p.toString(Path::PATH_UNIX); // "/usr/include/stdlib.h"

	p.resolve(p2);
	s = p.toString(Path::PATH_UNIX); // "/opt/Poco/include/Poco.h"

	return 0;
}
```

# Special Directories And Files

- Poco::Path provides static methods to obtain the paths to system specific special directories or files:
  - std::string current()
    returns the path for the current working directory
  - std::string home()
    returns the path to the user's home directory
  - std::string temp()
    returns the path to the system's directory for temporary files
  - std::string null()
    returns the path to the system's null file/device
    (e.g., "/dev/null" or "NUL:")

```
#include "Poco/Path.h"
#include <iostream>

using Poco::Path;

int main(int argc, char** argv)
{
    std::cout
        << "cwd:  " << Path::current() << std::endl
        << "home: " << Path::home() << std::endl
        << "temp: " << Path::temp() << std::endl
        << "null: " << Path::null() << std::endl;

    return 0;
}
```

# Paths And Environment Variables

- Paths found in configuration files often contain environment variables. Environment variables in a path must be expanded before such a path can be passed to Poco::Path.

- std::string expand(const std::string& path)
  returns a copy of path with all environment variables expanded.
  The syntax of environment variables is system specific (e.g., $VAR on Unix, %VAR% on Windows).
  On Unix, also expands "~/" to the current user's home directory.

```
#include "Poco/Path.h"

using Poco::Path;

int main(int argc, char** argv)
{
    std::string config("%HOMEDRIVE%%HOMEPATH%\\config.ini");
//  std::string config("$HOME/config.ini");

    std::string expConfig(Path::expand(config));

    return 0;
}
```

# Filesystem Roots

- void listRoots(std::vector<std::string>& roots)
  fills the given vector of strings with the names of all mounted root filesystems. On Windows, the list will consist of the available drive letters. On OpenVMS, the list will contain the names of all mounted disks. On Unix, the list will consist of exactly one slash ("/").

# Finding Files

- bool find(const std::string& pathList,
  const std::string& name, Path& path)
  searches the file with the given name in the locations specified in pathList. The paths in pathList must be delimited by the platform's path separator (";" on Windows, ":" on Unix). A relative path may be given in name.
  If the file is found in one of the locations given in pathList, the absolute path of the file is stored in path, and true is returned. Otherwise, false is returned and path remains unchanged.

- There is a variant of this function taking iterators to a string vector instead of a path list.

```
#include "Poco/Path.h"
#include "Poco/Environment.h"

using Poco::Path;
using Poco::Environment;

int main(int argc, char** argv)
{
    std::string shellName("cmd.exe"); // Windows
//  std::string shellName("sh");      // Unix

    std::string path(Environment::get("PATH"));

    Path shellPath;
    bool found = Path::find(path, shellName, shellPath);
    std::string s(shellPath.toString());

    return 0;
}
```

# Working With Files

- POCO has support for working with a file's metadata only. To access the actual data in a file, use the file streams provided by the standard library.
- With POCO, you can find out whether a file or directory exists, is readable or writable, when it was created or modified or how big it is.
- You can also change some of a file's attributes, rename a file, copy a file, or delete a file.
- Finally, you can create empty files (in an atomic operation), and directories.

# The File Class

- All file-related operations are available in Poco::File.
- #include "Poco/File.h"
- To create a Poco::File, you need to supply a path. This path can be in a Poco::Path, or in a string (std::string or C style). You can also create an "empty" file, and set the path at a later time, with an assignment.
- Poco::File supports full value semantics, including all relational operators (==, !=, <, <=, >, >=). The relational operators are implemented as plain string comparisons using the file's path.

# Querying File Attributes

- bool exists() const
  returns true if the file exists, false otherwise

- bool canRead() const
  returns true if the file is readable (the user has sufficient privileges to read from the file), false otherwise

- bool canWrite() const
  returns true if the file is writeable (the user has sufficient privileges to write to the file), false otherwise

- bool canExecute() const
  returns true if the file is executable, false otherwise

# Querying File Attributes (cont'd)

- bool isFile() const
  returns true if the file is a regular file (and not a directory, or symbolic link), false otherwise
- bool isLink() const
  returns true if the file is a symbolic link, false otherwise
- bool isDirectory() const
  returns true if the file is a directory, false otherwise
- bool isDevice() const
  returns true if the file is a device, false otherwise

# Querying File Attributes (cont'd)

- bool isHidden() const
  returns true if the file has the hidden attribute set (on Windows), or its name starts with a dot (Unix)
- Poco::Timestamp created() const
  returns the date and time the file was created
- Poco::Timestamp getLastModified() const
  returns the date and time the file was accessed
- File::FileSize getSize() const
  returns the size of the file in bytes. On most systems, File::FileSize is an unsigned 64-bit integer.

# Modifying File Attributes

- void setLastModified(Poco::Timestamp dateTime)
  sets the access timestamp of the file
- void setSize(FileSize newSize)
  sets the size of the file in bytes, e.g. to truncate a file
- void setWritable(bool flag = true)
  Makes the file writeable (if flag == true), or read-only
  (if flag == false), by setting the appropriate flags in the filesystem
- void setReadOnly(bool flag)
  same as setWritable(!flag)

# Rename, Copy, Delete

- void copyTo(const std::string& path) const
  copies the file to the given path (which can be a directory)
- void moveTo(const std::string& path) const
  copies the file to the given path (which can be a directory) and then deletes the original file
- void renameTo(const std::string& path)
  renames the file
- void remove(bool recursive = false)
  deletes the file. If the file is a directory and recursive == true, recursively deletes all files and subdirectories in the directory.

# Creating Files and Directories

- bool createFile()
  creates a new, empty file in an atomic operation. Returns true if the file has been created, or false if the file already exists. Throws a Poco::FileException if creation fails.

- bool createDirectory()
  creates a new directory. Returns true if the directory has been created, or false if the directory already exists. Throws a Poco::FileException if creation fails (e.g., if the parent directory does not exist).

- void createDirectories()
  creates a directory, as well as all parent directories, if necessary

# Reading a Directory

- void list(std::vector<std::string>& files) const
  void list(std::vector<File>& files) const
  fills the given vector with the names of all files in the directory. Internally, this uses a Poco::DirectoryIterator, which will be discussed shortly.

```
#include "Poco/File.h"
#include "Poco/Path.h"
#include <iostream>

using Poco::File;
using Poco::Path;

int main(int argc, char** argv)
{
    std::string tmpPath(Path::temp());
    tmpPath.pushDirectory("PocoFileSample");

    File tmpDir(tmpPath);
    tmpDir.createDirectories();

    bool exists   = tmpDir.exists();
    bool isFile   = tmpDir.isFile();
    bool isDir    = tmpDir.isDirectory();
    bool canRead  = tmpDir.canRead();
    bool canWrite = tmpDir.canWrite();
```

```
File tmpFile(Path(tmpPath, std::string("PocoFileSample.dat")));
if (tmpFile.createFile())
{
    tmpFile.setSize(10000);

    File tmpFile2(Path(tmpPath, std::string("PocoFileSample2.dat")));
    tmpFile.copyTo(tmpFile2.path());

    Poco::Timestamp now;
    tmpFile.setLastModified(now);

    tmpFile.setReadOnly();
    canWrite = tmpFile.canWrite();
    tmpFile.setWriteable();
    canWrite = tmpFile.canWrite();
}
```

```
        std::vector<std::string> files;
        tmpDir.list(files);
        std::vector<std::string>::iterator it = files.begin();
        for (; it != files.end(); ++it)
        {
            std::cout << *it << std::endl;
        }

        tmpDir.remove(true);

        return 0;
    }
```

# The DirectoryIterator Class

- Poco::DirectoryIterator provides an iterator-style interface for reading the contents of a directory.
- #include "Poco/DirectoryIterator.h"
- Poco::DirectoryIterator has some limitations:
  - only forward iteration (++) is supported
  - an iterator copied from another one will always point to the same file as the original iterator, even if the original iterator has been advanced (all copies share the same state)
- Poco::DirectoryIterator maintains a Poco::File and an absolute Poco::Path for the current item.

```
#include "Poco/DirectoryIterator.h"
#include <iostream>

using Poco::DirectoryIterator;
using Poco::Path;

int main(int argc, char** argv)
{
    std::string cwd(Path::current());
    DirectoryIterator it(cwd);
    DirectoryIterator end;
    while (it != end)
    {
        std::cout << it.name();
        if (it->isFile())
            std::cout << it->getSize();
        std::cout << std::endl;
        Path p(it.path());
        ++it;
    }

    return 0;
}
```

# The Glob Class

- Poco::Glob implements glob-style pattern matching as known from Unix shells.
- #include "Poco/Glob.h"
- In a pattern, '*' matches any sequence of characters, '?' matches any single character, [SET] matches any single character in the specified set, and [!SET] matches any single character not in the specified set.
  A set is composed of characters or ranges, e.g. [123] matches a digit 1, 2 or 3; [a-zA-Z] matches any lower- or uppercase letter.
  Special characters can be escaped with a backslash.

# The Glob Class (cont'd)

- A Poco::Glob is constructed with a pattern, and an optional option flag. The GLOB_DOT_SPECIAL option can be set to hide files starting with a period, in old Unix tradition.
- bool match(const std::string& subject)
  returns true if the path in subject matches the Glob's pattern, false otherwise.
- void glob(const std::string& pattern,
  std::set<std::string>& files, int options = 0)
  void glob(const Path& pattern,
  std::set<std::string>& files, int options = 0)
  fills the set with all files matching the given pattern.

```
#include "Poco/Glob.h"
#include <iostream>

using Poco::Glob;

int main(int argc, char** argv)
{
	std::set<std::string> files;
	Glob::glob("%WINDIR%\\system32\\*.exe", files);
//	Glob::glob("/usr/include/*/*.h", files);

	std::set<std::string>::iterator it = files.begin();
	for (; it != files.end(); ++it)
	{
		std::cout << *it << std::endl;
	}

	return 0;
}
```

# Temporary Files

- Many programs need temporary files, which can be characterized as follows:
  - a temporary file is created in a special system-specific directory (e.g., "/tmp/" on Unix systems)
  - a temporary file has an automatically generated unique name
  - a temporary file must be deleted when it is no longer needed
- Poco::TemporaryFile helps in doing all these things.
- #include "Poco/TemporaryFile.h"

# The TemporaryFile Class

- Poco::TemporaryFile is derived from Poco::File.
- The constructor automatically creates a unique file name, placed in the system-specific directory for temporary files. The file itself is not created.
- The destructor deletes the file, if it has been created.
- Alternatively, deletion can be postponed until the program terminates, or disabled altogether.
- Arbitrary files can be registered for deletion upon termination of the program.

# TemporaryFile Functions

- void keep()
  disables automatic deletion of the file by the destructor
- void keepUntilExit()
  disables automatic deletion of the file by the destructor, and registers the file for deletion upon termination of the program
- static void registerForDeletion(const std::string& path)
  registers a file for automatic deletion upon termination of the program
- static std::string tempName()
  creates a unique path name for a temporary file

```
#include "Poco/TemporaryFile.h"
#include <fstream>

using Poco::TemporaryFile;

int main(int argc, char** argv)
{
    TemporaryFile tmp;
    std::ofstream ostr(tmp.path().c_str());
    ostr << "Hello, world!" << std::endl;
    ostr.close();

    return 0;
}
```

applied informatics



www.appinf.com | info@appinf.com
T +43 4253 32596 | F +43 4253 32096