Terminology

File system classes and other software are in namespace *std::filesystem*. The *filesystem* library (from C++ v. 17) is for file management (inspect what files are present, create, copy and delete them, etc.). It does not provide facilities for reading and writing.

A file in *filesystem* library may be:

- regular file, for example *data.txt*, *app.exe* etc.
- directory

A path is a sequence of elements identifying the location of existing or not yet created file in file system. The textual representation of a file is the pathname.

Paths can be absolute (unambiguously identifies the location, starts from the root) or relative (starts from some location). Generally, a path contains:

- root name (optional)
- root directory (optional)
- sequence of file names separated by directory separators
- regular file name (optrional)

Raw string literals

```
const char *pText1 = "Hello world";
const char *pText2 { "Hello world" };
char arr[]  { "Hello world" };
```

Here *"Hello world"* is a string constant or string literal. Remark that:

```
pText1[5] =  '\n'; // error
pText1[5] =  '\n'; // error
arr[5] =  '\n'; // correct
```

A string literal may contain escape sequences:

```
const char *pText3 = "He said \"Hello everybody!\"\nWe answered \"You are welcome!\"";
const char *pText4 = "C:\\Temp\\Data.txt";
```

In raw string literals escape sequences are not needed. A raw string literal starts with *R" (* and ends with *)"*.

```
const char *pText5 = R"(He said "Hello everybody!"
We answered "You are welcome!")";
const char *pText6 = R"(C:\Temp\Data.txt)";
```

If you have embedded an escape sequence into raw string, it is processed as it is in text:

```
cout << R"(C:\\Temp)" << endl; // prints C:\\Temp
```

For text containing *)"* extended raw string literal syntax is needed, see
https://en.cppreference.com/w/cpp/language/string_literal.html

Path (1)

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

Here we deal only with the most used methods of class *path*. The full description is on page https://en.cppreference.com/w/cpp/filesystem/path.html

Constructors:

```
path p1 { };  // empty path
path p2 ("c:\\temp\\data.txt");  // path to file c:\temp\data.txt
path p3("c:/temp/data.txt"); // the same, Windows accepts both slashes
path p4(R"(c:\temp\data.txt)"); // the same, raw string literal is used
path p5 { "c:\\temp\\data.txt" };
path p6 {"c:/temp/data.txt" };
path p7 {R"(c:\temp\data.txt)" };
string s1 ("c:\\temp\\data.txt");
wstring ws1 (L"c:\\temp\\data.txt");
path p8(s1);  // pathname is a string object
path p8 (ws1);
path p10(R"(C:\Projects\University\IAX0587\Slides)");  // path to directory Slides
path p11 = p2;  // copy constructor
```

Path (2)

If we have got a path, we must at first check what is it actually:

```
int main(int arg c, char *argv[])
{ // suppose that the first command line argument specifies the input data file
  if (argc < 2) {
       cout << "Command line arguments missing" << endl;
       return 1;
    }
 path p { argv[1] };
 if (is_regular_file(p))
     cout << "Input data file found" << endl;
 else if (is_directory(p)) {
     cout <<  "Command line argument specifies a directory" << endl;
     return 1;
}
else if (exists(p)) {
     cout <<  "Command line argument specifies a special file" << endl;
     return 1;
}
else {
     cout <<  "Path specified by the ommand line argument does not exist" << endl;
     return 1;
}
```

Path (3)

Rather often, more checking is needed, let us continue the example:

```
if (file_size(p) == 0) {
    cout <<  "Input data file is empty" << endl;
    return 1;
}
if (!p.is_absolute()) {
      cout << "Relative path to input data is not allowed" << endl;
      return 1;
}
if (tolower(p.root_name().string().at(0)) != 'c') {
  // method root_name() of class path returns path, in our example path("C:")
  // method string() of class path returns the pathname as string
      cout << "Wrong disk name" << endl;
      return 1;
}
if (p.extension().string() != ".txt) {
 // method extension() of class path returns path, in our example path(".txt")
     cout << "Input data must be a .txt file" << endl;
     return 1;
}
```

Path (4)

A path may be in generic format (in that case it is portable to computers running on other platforms) or in native format (specific to the underlying system). For operating systems following the POSIX standard (Linux, macOS) and Windows those two formats are identical.

Although we mostly define a path with C-string or string object, the path is not a string:

```
path p ("c:\\temp\\data.txt");
string s = p.string();
wstring ws = p.wstring();
const wchar_t *pc = p.c_str();
cout << p << endl; // prints "c:\\temp\\data.txt"
```

There are several methods to inspect the path:

```
bool b1 = p.empty();
bool b2 = p.is_absolute();
bool b3 = p. has_filename(); // i.e. is not a directory, the same as is_regular_file(p)
```

To decomposite a path:

```
path p1 = p.root_name();  // here path("c:")
path p2 = p.parent_path(); // here path("c:\temp")
path p3 = p.relative_path(); // here path("temp\data.txt")
path p4 = p.filename();  // complete name including the extension, here path("data.txt")
path p5 = p.stem();  // filename without extension, here path "data"
path p6 = p.extension(); // here path(".txt")
```

Path (5)

Iteration through path is also possible:

```
path test { "c:/temp/data/test1.txt" };
for (path p: test) {
    cout << p << ' ';  // prints "c:" "/" "temp" "data" "test1.txt"
}
```

Some methods for path modifications:

```
path p { "c:/temp" };
p.append("data.txt");
cout << p << endl;  // prints "c:/temp\\data.txt"
cout << p.string() << endl;  // prints c:/temp\data.txt (different formats)
p.replace_filename("testdata.txt");
cout << p.string() << endl; // prints c:/temp\testdata.txt
p.replace_extension("bin");
cout << p.string() << endl; // prints c:/temp\testdata.bin
p.remove_filename();
cout << p.string() << endl; // prints c:/temp\
```

Current path

Method *current_path* returns the absolute path of the current working directory. If you have created a Visual Studio project in folder *C:\Projects\Coursework*, (i.e. the solution file is *C:\Projects\Coursework\Coursework.sln*):

```
cout << current_path().string() << endl;  //prints C:\Projects\Coursework\Coursework
```

The executable, however, is in folder *C:\Projects\Coursework\x64\Debug*. Generally, the current working directory may or may not be the directory in which the executable is located.

The problem is that when you have delivered the release of your application, the user may insert it into any folder. You application rather often needs additional auxiliary files for input data. Mostly, those files must be in the same folder where ithe executable is located. So, to open them the application needs to know not the current working directory but the directory containing the executable itself and its auxiliary files.

There are no tools in filesystem library to retrieve the executable path. The solution depends on the undelying operating system. In Windows 11 the following code worked (put it into a separate *.cpp file):

```
#define WIN32_LEAN_AND_MEAN
#include "Windows.h"
char Buf[2048];
GetModuleFileName(NULL, Buf, 2048);
path p(Buf);
cout << p.string() << endl; // prints C:\Projects\Coursework\x64\Debug\Coursework.exe
```

File attributes (1)

Some functions (they are not class methods) to find the file attributes:

```
path p ("c:\\temp\\data.txt");
bool b1 = exists(p);
bool b2 = is_regular_file(p);
bool b3 = is_directory(p);
bool b4 = is_empty(p);  // for files and directories
uintmax_t = file_size(p);  // max width unsigned integer, i.e. unsigned long long
```

The file can be resized. If the size was increased, the modified file has zeroes at the end.

```
resize_file(p, 1024UL); // now the size of file is 1 kB
```

It is possible to get information about the file system containing the path:

```
cout << space(p).capacity << endl;  // total number of bytes
cout << space(p).free << endl;  // total number of free bytes
```

All the functions presented above may throw exception (for example in case of incorrect input data), therefore the call should be put into try-catch block. For example:

```
bool b;
try {
    b = is_empty(p);
}
catch (exeption &e) {
    cout << e.what();  << endl;
}
```

File attributes (2)

However, if the work of application is not controlled by human operator and the application must make decisions automatically, printing of error messages is not an acceptable solution.

Therefore all the methods listed on the previous slide have two overloads, for example

```
bool is_empty(path p);
bool is_empty(path p, error_code ec) noexcept;
```

About *std::error_code* class see https://en.cppreference.com/w/cpp/error/error_code.html. Enumeration class *errc (*see https://en.cppreference.com/w/cpp/error/errc.html) presents the full list of possible errors.

```
#include <system_error>
error_code ec;
path p ("c:\\temp\\data.txt");
bool b = is_empty(p, ec);
if (ec) {   // error has occurred, value of b has no meaning
    if (ec == errc::no_such_file_or_directory) {
          ……. // here a standard implicit cast is provided
     }
     else if {ec == …….)
        ……
         ……...
}
```

The *error_code* class also has method *message()*, returning a string.

File management (1)

Some functions (they are not class methods) to manage regular files and directories:

```
path p1 { "c:\\temp\\tests\\" };
bool b;
try {  // https://en.cppreference.com/w/cpp/filesystem/create_directory.html
      b = create_directory(p1); // creates an empty directory
                                // if the directory already exists, returns false, but does not
                                // throw exception
}
catch (exception &e) {
      cout << e.what() << endl;
}
```

Here method *create_directory()* works successfully only if directory *c:\temp* already exists.

```
path p2 { "c:\\temp\\tests\\data" };
try {
     b = create_directories(p2);
}
catch (exception &e) {
      cout << e.what() << endl;
}
```

Here method *create_directories()*, if necessary, creates also directories *c:\temp* and *c:\temp\tests*.

File management (2)

Alternative solution:

```
path p1 { "c:\\temp\\tests\\data" };  // here c:\temp\tests does not exist
bool b = create_directory(p1, ec);
if (ec) {
    cout << ec.message() << endl;  // prints The system cannot find the path specified
}
```

or

```
if (ec) {
    if (ec == errc::no_such_file_or_directory) {
          ……….
     }
     else if {ec == errc::not_a_directory) {
          ……….
     }
     ……….
}
```

Similarly:

```
create_directories(p1, ec);
if (ec) {
    ……….
}
```

File management (3)

To create files use methods from fstream library (see slides from IAX0586).

```
path p1 { "c:\\temp\\test\\data1.txt" }, p2 { "c:\\temp\\test" }, p3 { "c:\\temp" };
bool b1, b2, b3;
uintmax_t = n;
try { // https://en.cppreference.com/w/cpp/filesystem/remove.html
    b1 = remove(p1);  // deletes file data1.txt, returns false if the file was not found
                      // (it is not an error)
    b2 = remove(p2);  // deletes empty directory c:\temp\test, throws exception if
                      // the directory is not empty
    n = remove_all(p3); // deletes directory c:\temp and all its subdirectories
                        // together with files inside them, returns the number of deleted files
                        // and directories
}
catch (exception &e) {
    cout << e.what() << endl;
}
```

Alternative:

```
error_code ec;
b1 = remove(p1, ec);
n = remove_all(p3, ec);
```

File management (4)

```
path p1 { "c:\\temp\\data1.txt" }, p2 { "c:\\temp"\\data2.txt };
bool b;
try {  // https://en.cppreference.com/w/cpp/filesystem/copy_file.html
       b = copy_file(p1, p2, copy_options::overwrite_existing);
}
catch (exception &e) {
        cout << e.what() << endl;
}
```

Here method *copy_file()* copies *data1* into *data2*. If *data2* exists, it will be overwritten. In case of *copy_options:: skip_existing* an existing file is not overwritten and copying fails. In case of *copy_options::update_existing* an existing file is overwritten only if the original file (here specified by path *p1*) is newer. If no errorrs occurred, the return value shows was the copying provided or not.

Copy options: see https://en.cppreference.com/w/cpp/filesystem/copy_options.html

As the other file management functions, *copy_file(()* method has two overloads:

```
bool copy_file(path p1, path p2, copy_options co);
bool copy_file(path p1, path p2, copy_options co, error_code ec);
```

The both overloads may throw exceptions.

File management (5)

```
path p1 { "c:\\temp" }, p2 { "c:\\temp1" };
try {  // https://en.cppreference.com/w/cpp/filesystem/copy.html
       copy(p1, p2, copy_options::recursive);
}
catch (exception &e) {
       cout << e.what() << endl;
}
```

Here *c:\temp* with all its files and subdirectories are copied into *c:\temp1*. Method *copy()* copies also files, for example:

```
path p3 { "c:\\temp\\data1.txt" }, p4 { "c:\\temp"\\data2.txt };
try {
       copy(p3, p4, copy_options::skip_existing);
}
catch (exception &e) {
       cout << e.what() << endl;
}
```

Method *copy()* has no output value. The both overloads can throw exceptions*:*

```
void copy(path p1, path p2, copy_options co);
void copy(path p1, path p2, copy_options co, error_code ec);
```

File management (6)

```
path p1 { "c:\\temp\\tests\\data" }, p2 { "c:\\temp\\tests\\old_data" };
try {  // https://en.cppreference.com/w/cpp/filesystem/rename.html
        rename(p1, p2);
}
catch (exception &e) {
         cout << e.what() << endl;
}
```

Here directory *c:\temp\tests\data* is renamed to *c:\temp\tests\old_data*.

```
path p3 { "c:\\temp\\tests\\old_data" }, p4 { "c:\\experiments" };
try {
        rename(p3, p4);
}
catch (exception &e) {
         cout << e.what() << endl;
}
```

Here *old_data* is deleted and its contents copied into new directory *experiments*. Method *rename()* may just rename, move to another location or do the both. Overloads are:

```
void rename(path p1, path p2);
void rename(path p1, path p2, error_code ec); noexept
```

Iterating over directories

```
path p { "C:\\Projects\\Coursework" };
directory_iterator dit { p };
for (const directory_entry& entry : dit)
{ // https://en.cppreference.com/w/cpp/filesystem/directory_iterator.html
  // https://en.cppreference.com/w/cpp/filesystem/directory_entry.html
  cout << entry.path().string() << endl;
}
```

Prints all the files and subdirectories in *C:\Projects\Coursework*, but not the contents of subdirectories

```
recursive_directory_iterator rdit{ p };
for (const directory_entry& entry : rdit)
{  // https://en.cppreference.com/w/cpp/filesystem/recursive_directory_iterator.html
   cout << entry.path().string() << endl;
}
```

Prints the complete tree, i.e. all the files in all the subdirectories. Class *directory_entry* has several methods analogous to methods from slide *File attributes (1)*. Example:

```
for (const directory_entry& entry : rdit) {
   if entry.is_regular_file() {   // prints only the files and their lengths
       cout << entry.path().string() << ' ' << entry.file_size() << " bytes" << endl;
   }
}
```