BEGINNER SERIES / FIRST EDITION

C++ Automation Lab

Part III

Files, Folders and Safe Previews

200 pages of lessons, runnable code, worked cases, debugging, tests, and guided project space.

Six small programs / 24 practice cases / Companion source files

This is a hands-on workbook. Its 200 pages include generous writing and planning space. It is not 200 pages of continuous reference prose.

READ FIRST

Use the lab, one case at a time

Read each unit lesson, build its source, and run the four cases. Cases reuse the same complete program with different inputs so you can study boundaries and failure modes. Each case occupies eight workbook pages.

All source files are included separately. Copying from the PDF is optional. Compare both output and process status, and preserve the original fixture before extending a program.

Part I starts from zero. Later volumes assume earlier skills. The main path uses C++17, a local terminal, and trusted ASCII lab paths. No external C++ libraries are required.

Examples are deliberately bounded teaching tools. They do not implement desktop mouse recording, browser automation, service deployment, or a complete backup system.

WINDOWS / VS CODE

Install and verify the toolchain

Install VS Code and its Microsoft C/C++ extension. Install a compiler separately. The official VS Code MinGW guide describes MSYS2 and its UCRT64 GCC toolchain. Use that guide for current installation steps.

```
g++ --version
g++ -std=c++17 -Wall -Wextra -pedantic example.cpp -o example.exe
.\example.exe
```

Replace example.cpp with the supplied source name. Run the command in its sources folder. If g++ is not recognized, inspect the toolchain installation and PATH, then open a new terminal. An editor extension alone does not install GCC.

Alternative for MSVC Developer PowerShell

```
cl /std:c++17 /EHsc /W4 example.cpp /Fe:example.exe
```

Use a dedicated lab directory. A relative file path resolves from the process working directory, which is not always the executable directory.

OFFICIAL DOCUMENTATION

References and work record

VS Code C++ setup

https://code.visualstudio.com/docs/cpp/config-mingw

C++ filesystem

https://learn.microsoft.com/cpp/standard-library/filesystem

File and string streams

https://learn.microsoft.com/cpp/standard-library/fstream

Numeric conversion

https://learn.microsoft.com/cpp/standard-library/charconv-functions

Task Scheduler

https://learn.microsoft.com/windows/win32/taskschd/task-scheduler-start-page

Documentation checked 14 November 2026. Tool setup can change. Open official sources for the current version.

Record your compiler version and lab folder.

UNIT 1 / CASE 1 / TEXT FILE / LEARN

Path Parts

A path is structured data. Build and inspect it through filesystem operations instead of manually splitting on slashes. These examples use relative ASCII paths so their printed components remain portable.

Case focus / Text file

extension returns only the final suffix including its dot. filename is a lexical operation and does not require the file to exist. Windows narrow argv is not a complete Unicode path solution; use ASCII lab paths.

Before running, predict one output and one possible failure.

UNIT 1 / CASE 1 / TEXT FILE / SET UP

Prepare the fixture

Source file / path_parts.cpp

Build from the sources folder, then use the input contract below. The companion cases.json contains the exact test data and expected output. Escape notation in that file represents actual newline and tab bytes.

```
g++ -std=c++17 -Wall -Wextra -pedantic path_parts.cpp
  -o path_parts.exe
```

The command above wraps for reading. Enter it on one terminal line.

Pass one quoted argument 'notes/today.txt' (or no arguments if empty).

Fixture notation / '@args:notes/today.txt'

Write your exact invocation and working directory.

UNIT 1 / CASE 1 / TEXT FILE / BUILD

Complete runnable source

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

int main(int argc, char* argv[]) {
    if (argc != 2) return 2;
    fs::path p(argv[1]);
    std::cout << "name=" << p.filename().generic_string() << '\n';
    std::cout << "stem=" << p.stem().generic_string() << '\n';
    std::cout << "extension=" << p.extension().generic_string() << '\n';
}
```

UNIT 1 / CASE 1 / TEXT FILE / OBSERVE

Trace the result

Expected standard output

```
name=today.txt
stem=today
extension=.txt
```

Expected process status / 0. No diagnostic output is expected for this fixture.

extension returns only the final suffix including its dot. filename is a lexical operation and does not require the file to exist. Windows narrow argv is not a complete Unicode path solution; use ASCII lab paths.

Trace the important variables after each input record.

UNIT 1 / CASE 1 / TEXT FILE / DEBUG

Find and repair a defect

Assume stem removes every suffix.

Work on a copy of the source. Predict the symptom before editing. A deliberately broken program can fail compilation, produce the wrong result, or fail to terminate. Stop it if necessary.

Repair explanation

For archive.tar.gz, stem is archive.tar. Remove multiple suffixes only if your explicit policy requires it.

Record the symptom, the smallest repair, and the case that proves it.

UNIT 1 / CASE 1 / TEXT FILE / TEST

Verify the contract

Run this case from a fresh fixture. Compare standard output exactly, then record status. Whitespace, ordering, and empty results are part of the contract.

Repeat with another case from the same unit. Choose one boundary that has not yet been exercised. Test an invalid input appropriate to this program, and write the intended behavior before changing code.

Do not assume every arbitrary invalid input is already rejected by these teaching examples. Record limitations openly. Use Part IV and Part V validation techniques to strengthen a parser.

Input | Expected output | Expected status | Actual result

UNIT 1 / CASE 1 / TEXT FILE / PRACTICE

Extend with one requirement

Print the parent path and compare a root path, a relative path, and a trailing separator.

Preserve the passing baseline. State the new input and output contract, choose one normal case and one boundary, then change the smallest useful unit of code.

If your change introduces writing, keep it inside a disposable lab directory, preview the intended operation, and define how an existing destination is handled. Never use a personal document folder as a fixture.

Write a small design, acceptance cases, and the final command.

UNIT 1 / CASE 1 / TEXT FILE / REVIEW

Explain and hand off

Explain the tool to a beginner in your own words. Name the accepted input, observable output, status meanings, and one limitation. Include the compiler version and the folder from which the tool should run.

Keep source, fixtures, expected results, and a brief run note together. A repeatable small tool is easier to maintain than a clever command with undocumented assumptions.

What would another reader need to reproduce your result?