DNS-Resolver-with-Poco-C-libraries

This project implements a command-line DNS resolver in C++ using the Poco Net library. It takes a domain name as input and returns its corresponding IP address. The resolver features a simple caching mechanism to improve performance for repeated queries and includes error handling for invalid domains. The project demonstrates practical use of networking libraries, CMake build system, and modern C++ practices.
This project involves the design and implementation of a DNS Resolver in C++ using the Poco C++ Libraries. The resolver is capable of efficiently converting domain names into IP addresses. To enhance performance and accuracy, the system also supports caching of resolved IPs and recursive resolution of domain names. The development was structured over five days, each with clearly defined goals and deliverables to ensure progressive enhancement and modular implementation.

Day 1: Project Setup and DNS Basics
The first day focused on laying the foundation for the DNS Resolver project. The development environment was configured with Visual Studio Code and the MSYS2 MinGW UCRT64 toolchain. The Poco C++ libraries were installed and linked to support network programming features, particularly DNS query capabilities. A basic project structure was created, including folders for source files (src), headers (include), and build configuration (CMakeLists.txt). Alongside the setup, an in-depth study of DNS fundamentals was conducted, covering concepts like domain names, IP addressing, DNS hierarchy, and the resolution process. The architectural blueprint for the resolver was drafted, outlining separate components for query handling, caching, recursive resolution, and main execution flow.

Day 2: Implementing DNS Query Handling
On the second day, the implementation of core DNS query handling was carried out using the Poco::Net module. A function was developed to send DNS queries and retrieve corresponding IP addresses. Using Poco::Net::DNS::hostByName(), the resolver was able to perform synchronous DNS lookups. Additionally, logic was added to parse and display the returned IP addresses to the user. These implementations were tested with several sample domains such as example.com, google.com, and others to verify correctness. This phase ensured that the resolver could interact with external DNS infrastructure and extract meaningful data for subsequent operations.

Day 3: Caching Resolved Addresses
To improve efficiency and minimize redundant network calls, a caching layer was introduced on the third day. The cache was implemented using standard C++ containers such as std::unordered_map, mapping domain names to resolved IP addresses and timestamps. The logic was designed to first check the cache before initiating any new DNS queries. This approach ensured that repeated queries to the same domain would return results instantly from memory, reducing DNS traffic and latency. The caching mechanism was successfully

tested by running multiple repeated queries, validating that only the first resulted in an actual DNS request while others were served from the cache.

Day 4: Recursive Resolution
The fourth day focused on enabling recursive resolution capabilities, allowing the resolver to handle multi-level domain queries that require interaction with several authoritative DNS servers. A recursive resolver class was implemented that mimicked the behavior of a recursive DNS server, resolving a domain name by following a hierarchy of name servers from the root to the authoritative server. This was done by simulating recursive lookup chains using the Poco DNS tools and enhancing the system with robust error handling. Mechanisms were put in place to handle timeouts, invalid domain formats, and unreachable DNS servers gracefully. Extensive testing ensured that complex domains like sub.domain.example.com were resolved effectively.

Day 5: Testing, Documentation, and Optimization
The final day involved a complete evaluation of the DNS resolver’s performance and functionality. Test cases were written to simulate both normal and edge-case scenarios, including empty inputs, repeated queries (to test caching), and deep recursive lookups. The codebase was reviewed and optimized for performance, focusing on reducing memory usage and improving response times. The resolver’s logic was modularized and commented clearly for maintainability. Documentation was prepared, covering setup instructions, architecture overview, usage examples, and future improvement suggestions. Additionally, a summary presentation highlighting major design decisions, challenges like parsing DNS responses and recursive querying logic, and the strategies adopted for optimization was created.

Installing and Setting up MSYS2 (MinGW UCRT64 Toolchain):

Download and install MSYS2 from https://www.msys2.org/
Open the terminal:
Start → MSYS2 UCRT64 (Important: use UCRT64, not the default shell)
Update MSYS2 packages (run twice):
pacman -Syu
Close and reopen terminal
pacman -Syu

Install build tools and compiler:
pacman -S --needed base-devel mingw-w64-ucrt-x86_64-toolchain

Install CMake and Git:
pacman -S mingw-w64-ucrt-x86_64-cmake mingw-w64-ucrt-x86_64-git

Install Poco C++ Libraries:

```
pacman -S mingw-w64-ucrt-x86_64-poco
```

Add UCRT64 to your PATH (if not already):

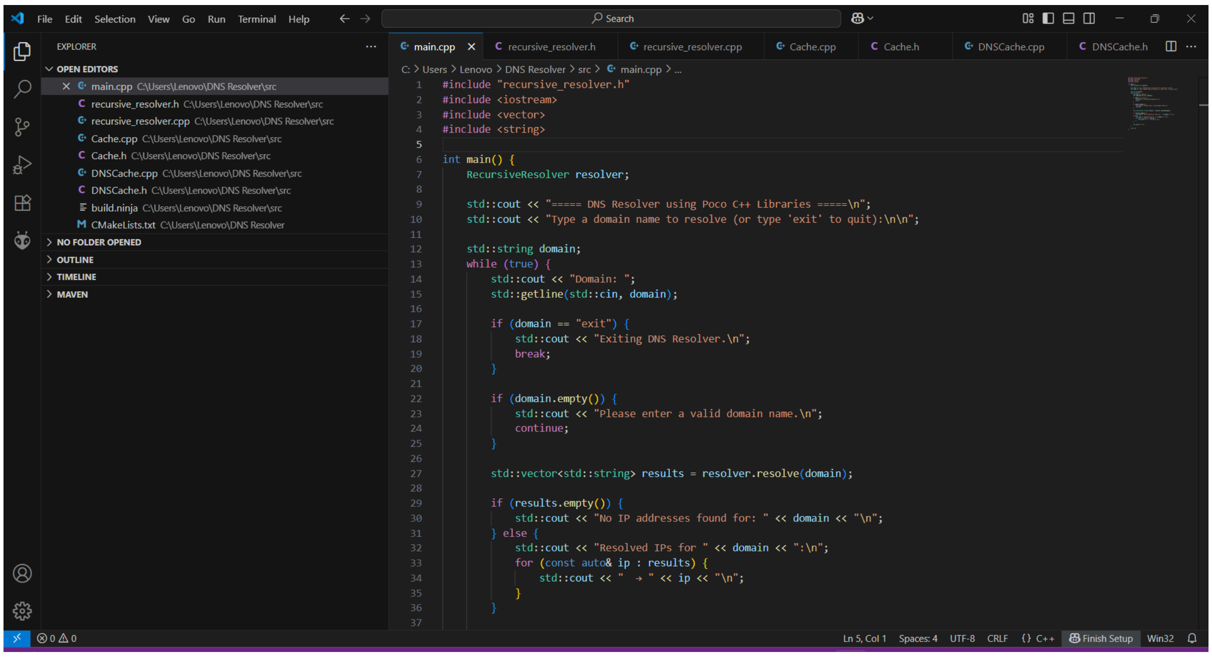
```
export PATH="/ucrt64/bin:$PATH"
```

In brief,

- Open terminal
  Start → MSYS2 UCRT64
-Update packages
  ```
  pacman -Syu (twice)
  ```
- Install compiler/toolchain
  ```
  pacman -S --needed base-devel mingw-w64-ucrt-x86_64-toolchain
  ```
- Install Poco
  ```
  pacman -S mingw-w64-ucrt-x86_64-poco
  ```
- Install CMake
  ```
  pacman -S mingw-w64-ucrt-x86_64-cmake
  ```
- Install Git
  ```
  pacman -S mingw-w64-ucrt-x86_64-git
  ```
- Build using CMake
  ```
  cmake .. && cmake --build .
  ```
- Run the program
  ```
  ./dns_resolver or ./build/dns_resolver.exe
  ```

Purpose	Command
Open terminal	Start → MSYS2 UCRT64
Update packages	*pacman -Syu (*twice*)*
Install compiler/toolchain	*pacman -S --needed base-devel mingw-w64-ucrt-x86_64-toolchain*
Install Poco	*pacman -S mingw-w64-ucrt-x86_64-poco*
Install CMake	*pacman -S mingw-w64-ucrt-x86_64-cmake*
Install Git	*pacman -S mingw-w64-ucrt-x86_64-git*
Build using CMake	*cmake .. && cmake --build .*
Run the program	*./dns_resolver or ./build/dns_resolver.exe*

Folder Created


File
Edit
Selection
View
Go
Run
Terminal
Help
Search
EXPLORER
OPEN EDITORS
main.cpp C:\Users\Lenovo\DNS Resolver\src
recursive_resolver.h C:\Users\Lenovo\DNS Resolver\src
recursive_resolver.cpp C:\Users\Lenovo\DNS Resolver\src
Cache.cpp C:\Users\Lenovo\DNS Resolver\src
Cache.h C:\Users\Lenovo\DNS Resolver\src
DNSCache.cpp C:\Users\Lenovo\DNS Resolver\src
DNSCache.h C:\Users\Lenovo\DNS Resolver\src
build.ninja C:\Users\Lenovo\DNS Resolver\src
CMakeLists.txt C:\Users\Lenovo\DNS Resolver
NO FOLDER OPENED
OUTLINE
TIMELINE
MAVEN
main.cpp
recursive_resolver.h
recursive_resolver.cpp
Cache.cpp
Cache.h
DNSCache.cpp
DNSCache.h
C: > Users > Lenovo > DNS Resolver > src > main.cpp > ...
Ln 5, Col 1
Spaces: 4
UTF-8
CRLF
{} C++
Finish Setup
Win32

```cpp
#include "recursive_resolver.h"
#include <iostream>
#include <vector>
#include <string>

int main() {
    RecursiveResolver resolver;

    std::cout << "===== DNS Resolver using Poco C++ Libraries =====\n";
    std::cout << "Type a domain name to resolve (or type 'exit' to quit):\n\n";

    std::string domain;
    while (true) {
        std::cout << "Domain: ";
        std::getline(std::cin, domain);

        if (domain == "exit") {
            std::cout << "Exiting DNS Resolver.\n";
            break;
        }

        if (domain.empty()) {
            std::cout << "Please enter a valid domain name.\n";
            continue;
        }

        std::vector<std::string> results = resolver.resolve(domain);

        if (results.empty()) {
            std::cout << "No IP addresses found for: " << domain << "\n";
        } else {
            std::cout << "Resolved IPs for " << domain << ":\n";
            for (const auto& ip : results) {
                std::cout << "  → " << ip << "\n";
            }
        }

```

1. Project Explorer (Left Panel)

Your folder: C:\Users\Lenovo\DNS Resolver\src\

It includes the following files:

File	Purpose
main.cpp	Entry point of the DNS Resolver. User input + interaction happens here.
recursive_resolver.h	Header file declaring the RecursiveResolver class.
recursive_resolver.cpp	Implements the recursive resolution logic. Uses Poco functions.
Cache.cpp	Source file to implement basic caching functionality.
Cache.h	Header file declaring caching class/structure.
DNSCache.cpp	Possibly extended caching with DNS-specific logic.
DNSCache.h	Header for DNSCache.cpp.
build.ninja	Ninja build system file (automatically generated if you used Ninja).

CMakeLists.txt	CMake build configuration file.

2. main.cpp (Line by Line)

This is your **driver program**, handling input/output and calling resolution logic.

```
#include "recursive_resolver.h"
#include <iostream>
#include <vector>
#include <string>
```

Includes the header for your resolver class and standard libraries:
- `iostream` → for `std::cout`/`std::cin`
- `vector` → to store multiple IP addresses
- `string` → to store domain names

```
int main() {
   RecursiveResolver resolver;
```

- Entry point of the program.
- Creates an object resolver of type RecursiveResolver which likely contains the DNS logic.

```
std::cout << "===== DNS Resolver using Poco C++ Libraries =====\n";

std::cout << "Type a domain name to resolve (or type 'exit' to quit):\n\n";
```

Greets the user and shows instructions.

```
   std::string domain;
   while (true) {
      std::cout << "Domain: ";
      std::getline(std::cin, domain);
```

Takes **user input** for the domain.
Uses getline to allow spaces (if any) in the input.

```
  if (domain == "exit") {
      std::cout << "Exiting DNS Resolver.\n";
      break;
   }
```

- Allows user to exit the loop by typing `exit`.

```
    if (domain.empty()) {
        std::cout << "Please enter a valid domain name.\n";
        continue;
    }
```

Input validation: skips empty input.

```
std::vector<std::string> results = resolver.resolve(domain);
```

- Calls the `resolve()` function in `RecursiveResolver` class.
- Stores the result (IP addresses) in `results`.

```
    if (results.empty()) {
        std::cout << "No IP addresses found for: " << domain << "\n";
    } else {
        std::cout << "Resolved IPs for " << domain << ":\n";
        for (const auto& ip : results) {
            std::cout << "  -> " << ip << "\n";
        }
    }
```

If no IPs are found, informs the user.
Otherwise, prints each resolved IP address.

Final Output:

```
===== DNS Resolver using Poco C++ Libraries =====

Type a domain name to resolve (or type 'exit' to quit):



Domain: google.com

Resolved IPs for google.com:

  -> 142.250.182.206
```

```
Domain: openai.com

Resolved IPs for openai.com:

  -> 104.19.195.29

  -> 104.19.194.29


Domain: invalid.domain.abcdef

No IP addresses found for: invalid.domain.abcdef


Domain:

Please enter a valid domain name.


Domain: exit

Exiting DNS Resolver.
```