Forge

1.0.0

Generated on Tue Apr 22 2025 15:43:30 for Forge by Doxygen 1.13.2

Tue Apr 22 2025 15:43:30

Chapter 1

Forge

1.0.1 Overview

Forge is a **modern** and **user-friendly build system** written in **C++20**. It is designed to simplify and optimize the build process for projects of any size. With its powerful features and intuitive design, Forge is the perfect tool for developers looking for a flexible and efficient build system.

1.0.2 Key Features

- **Decomposition of Build Configurations**:
  Forge allows you to split build configurations into multiple reusable files, making it easier to manage complex projects.
- **Python-Like Imports**:
  Supports Python-like imports, optimizing the parsing and processing phases of configurations.
- **HTML Build Results**:
  Generates **HTML outputs** of the build results, making them easy to read and understand.
- **Caching**:
  Implements caching mechanisms to avoid redundant builds, improving performance.
- **Parallel Processing**:
  Supports parallel execution of tasks, significantly speeding up the build process.
- **Ease of Use**:
  Designed to be simple and intuitive, making it easy to learn and use for developers of all skill levels.
- **Flexibility**:
  A powerful and flexible system that can be adapted to any project.

1.0.3 Why Choose Forge?

1.0.3.1 1. Readable Build Outputs

Forge generates visually appealing **HTML reports** for build results, making it easy to identify issues and understand the build process.

1.0.3.2 2. Optimized for Speed

With features like **caching** and **parallel processing**, Forge ensures that your builds are as fast and efficient as possible.

1.0.3.3 3. Modern Design

Built with **C++20**, Forge leverages the latest advancements in the language to provide a robust and modern build system.

1.0.4 How It Works

1. **Define Your Build Configurations**:
   Write your build configurations in a modular and reusable way.
2. **Run Forge**:
   Use Forge to parse and process your configurations.
3. **View Results**:
   Check the **HTML output** for a detailed summary of the build process.

1.0.5 Directory Structure

```
Directory structure:
 kronos-192081-forge/
     README.md
     argparse.hpp
     cache.hpp
     coderunner.hpp
     configure
     Doxyfile
     graph.hpp
     main.cpp
     Makefile
     parser.cpp
     parser.hpp
     tabulate.hpp
     doxygen-awesome-css/
        README.md
        Doxyfile
        doxygen-awesome-darkmode-toggle.js
        doxygen-awesome-fragment-copy-button.js
        doxygen-awesome-interactive-toc.js
        doxygen-awesome-paragraph-link.js
        doxygen-awesome-sidebar-only-darkmode-toggle.css
        doxygen-awesome-sidebar-only.css
        doxygen-awesome-tabs.js
        doxygen-awesome.css
        LICENSE
        Makefile
        package.json
        .gitignore
        .npmignore
        docs/
           customization.md
           extensions.md
           tricks.md
           img/
        doxygen-custom/
           custom-alternative.css
           custom.css
           header.html
           toggle-alternative-theme.js
        img/
        include/
           MyLibrary/
               example.hpp
               subclass-example.hpp
        .github/
            workflows/
                publish.yaml
     test/
        2>&1
        array.txt
        array_gen
        cd.py
        forgefile
        output.html
        output2.html
        output3.html
        sample.c
        sample_dep.c
     test2/
        Query-Optimiser/
            README.md
```

```
        call.py
        clean.forge
        deploy.py
        forge_output.html
        forgefile
        inp.txt
        lexer.forge
        Makefile
        output.txt
        output2.html
        output3.html
        requirements.txt
        res.txt
        test.l
        test.y
        test_res
        test_target_translator.cxx
        test_translator.cxx
        test_translator.h
        translator.forge
        .gitignore
        tmp/
           in.txt
           out.txt
        .streamlit/
            config.toml
test_files/
    sample.mk
    sample2.mk
    sample3.mk
```

1.0.6 Installation

1.0.6.1 Prerequisites

- **C++20 Compiler** (e.g., GCC 10+, Clang 10+)
- **Python 3** (for certain features like HTML generation)
- **SQLite3** (for caching)
- **OpenSSL** (for hashing)

1.0.6.2 Steps

1. Clone the repository:
   ```
   git clone https://github.com/Kronos-192081/Forge.git
   cd forge
   ```
2. Install Pre-requisites:
   ```
   ./configure
   ```
3. Build Forge:
   ```
   make
   ```
4. Install Forge:
   ```
   sudo make install
   ```

1.0.7 Contributing

We welcome contributions to Forge! To get started:

```
1. Fork the repository.
2. Create a new branch for your feature or bug fix.
3. Submit a pull request.
```

1.0.7.1 Acknowledgments

Special thanks to p-ranav for the libraries argparse and tabulate which has streamlined the development of Forge.

Chapter 2

Namespace Index

2.1 Namespace List

Here is a list of all namespaces with brief descriptions:

Chapter 3

Class Index

3.1 Class List

Here are the classes, structs, unions and interfaces with brief descriptions:

Chapter 4

File Index

4.1 File List

Here is a list of all files with brief descriptions:

Chapter 5

Namespace Documentation

5.1 std Namespace Reference

Specialization of the `std::hash` template for the `Node` struct.

Classes

- struct hash< Node >

5.1.1 Detailed Description

Specialization of the `std::hash` template for the `Node` struct.
This specialization allows `Node` objects to be used as keys in unordered containers such as `std::unordered_map` or `std::unordered_set`. It computes the hash value for a `Node` based on its `name` attribute.

Chapter 6

Class Documentation

6.1 Cache Class Reference

A class to manage a file cache using an SQLite database.
`#include <cache.hpp>`

Public Member Functions

- Cache ()

  Constructs a Cache object and initializes the SQLite database.
- ~Cache ()

  Destroys the Cache object and closes the SQLite database connection.
- void add (const std::string &file_addr)

  Adds or updates a file's hash in the cache.
- std::string find (const std::string &file_addr)

  Finds the hash of a file in the cache.
- bool check (const std::string &file_addr)

  Checks if a file's current hash matches the hash stored in the cache.

Private Member Functions

- std::string computeHash (const std::string &file_path)

  Computes the SHA256 hash of a file.
- void initializeDB ()

  Initializes the SQLite database and creates the required table if it does not exist.

Private Attributes

- sqlite3 * db

  Pointer to the SQLite database connection.
- std::string db_path

  Path to the SQLite database file.

6.1.1 Detailed Description

A class to manage a file cache using an SQLite database.
The Cache class provides functionality to store file hashes in an SQLite database, compute file hashes using SHA256, and check if a file's hash matches the stored hash.
Definition at line 18 of file cache.hpp.

6.1.2 Constructor & Destructor Documentation

6.1.2.1 Cache()

`Cache::Cache () [inline]`

Constructs a Cache object and initializes the SQLite database.
The database is created in the user's home directory as `.forgecache` if it does not already exist.

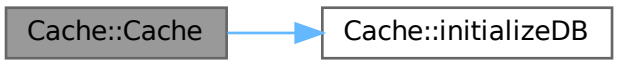
Exceptions

std::runtime_error	If the HOME environment variable is not set or the database cannot be opened.

Definition at line 85 of file cache.hpp.

```
00085             {
00086         const char* home = std::getenv("HOME");
00087         if (!home) {
00088             throw std::runtime_error("HOME environment variable not set");
00089         }
00090
00091         db_path = std::string(home) + "/.forgecache";
00092
00093         if (sqlite3_open(db_path.c_str(), &db) != SQLITE_OK) {
00094             throw std::runtime_error("Failed to open SQLite database");
00095         }
00096
00097         initializeDB();
00098     }
```

Here is the call graph for this function:



6.1.2.2 ~Cache()

`Cache::~Cache () [inline]`

Destroys the Cache object and closes the SQLite database connection.
Definition at line 103 of file cache.hpp.

```
00103             {
00104         if (db) {
00105             sqlite3_close(db);
00106         }
00107     }
```

6.1.3 Member Function Documentation

6.1.3.1 add()

```
void Cache::add (
            const std::string & file_addr)  [inline]
```

Adds or updates a file's hash in the cache.

Parameters

file_addr	The path to the file to be added or updated in the cache.

Exceptions

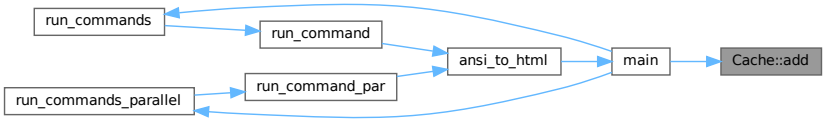
std::runtime_error	If the insert operation fails.

Definition at line 115 of file cache.hpp.

```
00115                                       {
00116         std::string file_hash = computeHash(file_addr);
00117
00118         const char* insert_query = R"(
00119             INSERT OR REPLACE INTO cache (file_addr, file_hash)
00120             VALUES (?, ?);
00121         )";
00122
00123         sqlite3_stmt* stmt;
00124         if (sqlite3_prepare_v2(db, insert_query, -1, &stmt, nullptr) != SQLITE_OK) {
00125             throw std::runtime_error("Failed to prepare insert statement");
00126         }
00127
00128         sqlite3_bind_text(stmt, 1, file_addr.c_str(), -1, SQLITE_STATIC);
00129         sqlite3_bind_text(stmt, 2, file_hash.c_str(), -1, SQLITE_STATIC);
00130
00131         if (sqlite3_step(stmt) != SQLITE_DONE) {
00132             sqlite3_finalize(stmt);
00133             throw std::runtime_error("Failed to execute insert statement");
00134         }
00135
00136         sqlite3_finalize(stmt);
00137     }
```

Here is the call graph for this function:

Here is the caller graph for this function:



6.1.3.2 check()

```
bool Cache::check (
            const std::string & file_addr)  [inline]
```

Checks if a file's current hash matches the hash stored in the cache.

Parameters

file_addr	The path to the file to be checked.

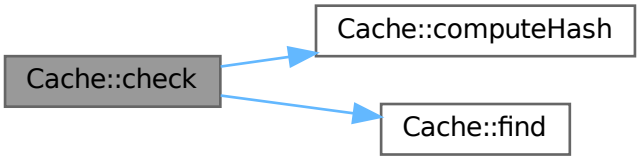
Returns

True if the file's current hash matches the stored hash, false otherwise.

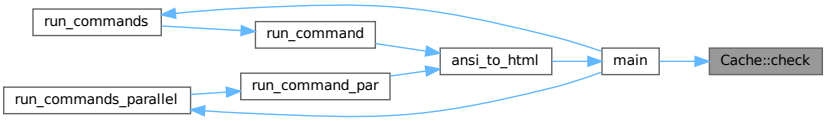
Definition at line 173 of file cache.hpp.

```
00173                                           {
00174         std::string file_hash = find(file_addr);
00175         if (file_hash.empty()) {
00176             return false;
00177         }
00178 
00179         std::string current_hash = computeHash(file_addr);
00180         return file_hash == current_hash;
00181     }
```

Here is the call graph for this function:



Here is the caller graph for this function:



6.1.3.3 computeHash()

```
std::string Cache::computeHash (
            const std::string & file_path)  [inline], [private]
```

Computes the SHA256 hash of a file.

Parameters

file_path	The path to the file whose hash is to be computed.

Returns

The computed SHA256 hash as a hexadecimal string.

Exceptions

std::runtime_error	If the file cannot be opened.

Definition at line 30 of file cache.hpp.

```
00030                                                   {
00031         std::ifstream file(file_path, std::ios::binary);
00032         if (!file.is_open()) {
00033             throw std::runtime_error("Unable to open file: " + file_path);
00034         }
00035
00036         SHA256_CTX sha256;
00037         SHA256_Init(&sha256);
00038
00039         char buffer[8192];
00040         while (file.read(buffer, sizeof(buffer))) {
00041             SHA256_Update(&sha256, buffer, file.gcount());
00042         }
00043         SHA256_Update(&sha256, buffer, file.gcount());
00044
00045         unsigned char hash[SHA256_DIGEST_LENGTH];
00046         SHA256_Final(hash, &sha256);
00047
00048         std::ostringstream oss;
00049         for (unsigned char c : hash) {
00050             oss << std::hex << std::setw(2) << std::setfill('0') << static_cast<int>(c);
00051         }
00052         return oss.str();
00053     }
```

Here is the caller graph for this function:

6.1.3.4 find()

```
std::string Cache::find (
            const std::string & file_addr)  [inline]
```

Finds the hash of a file in the cache.

Parameters

file_addr	The path to the file whose hash is to be retrieved.

Returns

The hash of the file as a string, or an empty string if the file is not in the cache.

Exceptions

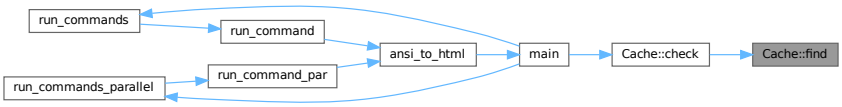
std::runtime_error	If the select operation fails.

Definition at line 146 of file cache.hpp.

```
00146                                                   {
00147         const char* select_query = R"(
00148             SELECT file_hash FROM cache WHERE file_addr = ?;
00149         )";
00150
00151         sqlite3_stmt* stmt;
00152         if (sqlite3_prepare_v2(db, select_query, -1, &stmt, nullptr) != SQLITE_OK) {
00153             throw std::runtime_error("Failed to prepare select statement");
00154         }
00155
00156         sqlite3_bind_text(stmt, 1, file_addr.c_str(), -1, SQLITE_STATIC);
00157
00158         std::string file_hash;
00159         if (sqlite3_step(stmt) == SQLITE_ROW) {
00160             file_hash = reinterpret_cast<const char*>(sqlite3_column_text(stmt, 0));
00161         }
00162
```

```
00163         sqlite3_finalize(stmt);
00164         return file_hash;
00165     }
```

Here is the caller graph for this function:



6.1.3.5 initializeDB()

`void Cache::initializeDB () [inline], [private]`

Initializes the SQLite database and creates the required table if it does not exist.

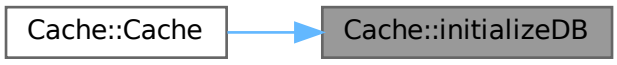
Exceptions

std::runtime_error	If the table creation fails.

Definition at line 60 of file cache.hpp.

```
00060                         {
00061         const char* create_table_query = R"(
00062             CREATE TABLE IF NOT EXISTS cache (
00063                 file_addr TEXT PRIMARY KEY,
00064                 file_hash TEXT
00065             );
00066         )";
00067 
00068         char* err_msg = nullptr;
00069         if (sqlite3_exec(db, create_table_query, nullptr, nullptr, &err_msg) != SQLITE_OK) {
00070             std::string error = "Failed to create table: ";
00071             error += err_msg;
00072             sqlite3_free(err_msg);
00073             throw std::runtime_error(error);
00074         }
00075     }
```

Here is the caller graph for this function:



6.1.4 Member Data Documentation

6.1.4.1 db

`sqlite3* Cache::db [private]`

Pointer to the SQLite database connection.

Definition at line 20 of file cache.hpp.

6.1.4.2 db_path

`std::string Cache::db_path [private]`

Path to the SQLite database file.

Definition at line 21 of file cache.hpp.
The documentation for this class was generated from the following file:

- cache.hpp

6.2 Graph< T > Class Template Reference

A generic directed graph implementation with utility functions.
`#include <graph.hpp>`

Public Member Functions

- void addNode (const T &node)

  Adds a node to the graph.
- void removeNode (const T &node)

  Removes a node from the graph.
- void addEdge (const T &from, const T &to)

  Adds a directed edge between two nodes.
- void removeEdge (const T &from, const T &to)

  Removes a directed edge between two nodes.
- std::optional< std::vector< T > > hasCycle ()

  Detects if the graph contains a cycle.
- std::vector< T > topologicalSort ()

  Performs a topological sort of the graph.
- int inDegree (const T &node)

  Computes the in-degree of a node.
- int outDegree (const T &node)

  Computes the out-degree of a node.
- void visualize (const std::string &filename, const std::string &imgfilename)

  Visualizes the graph by generating a DOT file and an image.

Private Member Functions

- bool dfsCycleDetection (const T &node, std::unordered_set< T > &visited, std::unordered_set< T > &recStack, std::vector< T > &path, std::vector< T > &cycle)

  Helper function for cycle detection using Depth-First Search (DFS).
- void dfsTopologicalSort (const T &node, std::unordered_set< T > &visited, std::stack< T > &Stack)

  Helper function for topological sorting using Depth-First Search (DFS).
- void generateDotFile (const std::string &filename)

  Generates a DOT file representing the graph.

Private Attributes

- std::unordered_map< T, std::unordered_set< T > > adjList

  The adjacency list representation of the graph. Maps each node to a set of its neighboring nodes.

6.2.1 Detailed Description

template<typename T>
class Graph< T >

A generic directed graph implementation with utility functions.
This class provides a representation of a directed graph using an adjacency list. It includes methods for adding and removing nodes and edges, detecting cycles, performing topological sorting, and visualizing the graph.

Template Parameters

T	The type of the nodes in the graph. Must be hashable and comparable.

Definition at line 23 of file graph.hpp.

6.2.2 Member Function Documentation

6.2.2.1 addEdge()

```
template<typename T>
void Graph< T >::addEdge (
            const T & from,
            const T & to)
```

Adds a directed edge between two nodes.

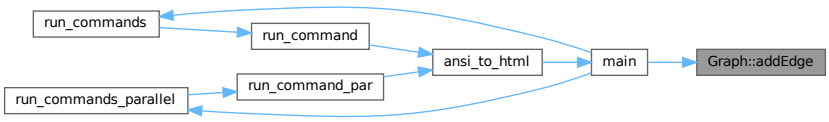
Parameters

from	The source node of the edge.
to	The destination node of the edge.

Definition at line 134 of file graph.hpp.

```
00134                                                {
00135     adjList[from].insert(to);
00136 }
```

Here is the caller graph for this function:



6.2.2.2 addNode()

```
template<typename T>
void Graph< T >::addNode (
            const T & node)
```

Adds a node to the graph.

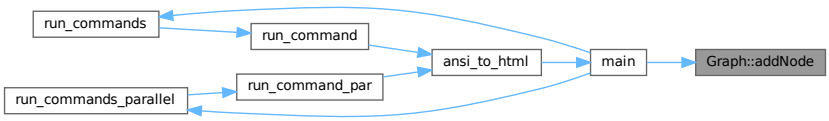
Parameters

node	The node to be added.

Definition at line 121 of file graph.hpp.

```
00121                                        {
00122     adjList[node];
00123 }
```

Here is the caller graph for this function:



6.2.2.3 dfsCycleDetection()

```
template<typename T>
bool Graph< T >::dfsCycleDetection (
            const T & node,
            std::unordered_set< T > & visited,
            std::unordered_set< T > & recStack,
            std::vector< T > & path,
            std::vector< T > & cycle)  [private]
```

Helper function for cycle detection using Depth-First Search (DFS).

Parameters

node	The current node being visited.
visited	A set of nodes that have been visited.
recStack	A set of nodes in the current recursion stack.
path	A vector to store the current path of nodes.
cycle	A vector to store the detected cycle if one exists.

Returns

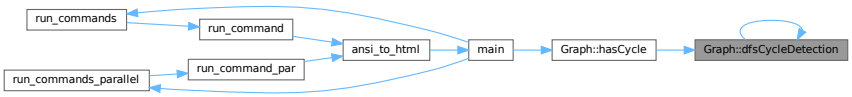
True if a cycle is detected, false otherwise.

Definition at line 158 of file graph.hpp.

```
00158
    {
00159     if (recStack.find(node) != recStack.end()) {
00160         auto it = std::find(path.begin(), path.end(), node);
00161         cycle.assign(it, path.end());
00162         return true;
00163     }
00164     if (visited.find(node) != visited.end()) {
00165         return false;
00166     }
00167     visited.insert(node);
00168     recStack.insert(node);
00169     path.push_back(node);
00170     for (const auto& neighbor : adjList[node]) {
00171         if (dfsCycleDetection(neighbor, visited, recStack, path, cycle)) {
00172             return true;
00173         }
00174     }
00175     recStack.erase(node);
00176     path.pop_back();
00177     return false;
00178 }
```

Here is the call graph for this function:

Here is the caller graph for this function:



6.2.2.4 dfsTopologicalSort()

```
template<typename T>
void Graph< T >::dfsTopologicalSort (
            const T & node,
            std::unordered_set< T > & visited,
            std::stack< T > & Stack)  [private]
```

Helper function for topological sorting using Depth-First Search (DFS).

Parameters

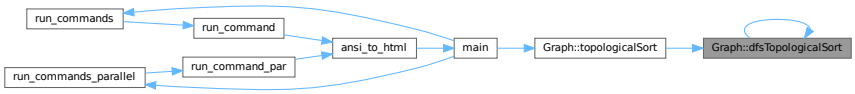
node	The current node being visited.
visited	A set of nodes that have been visited.
Stack	A stack to store the topologically sorted nodes.

Definition at line 198 of file graph.hpp.

```
00198                                                                                                {
00199     visited.insert(node);
00200     for (const auto& neighbor : adjList[node]) {
00201         if (visited.find(neighbor) == visited.end()) {
00202             dfsTopologicalSort(neighbor, visited, Stack);
00203         }
00204     }
00205     Stack.push(node);
00206 }
```

Here is the call graph for this function:

Here is the caller graph for this function:



6.2.2.5 generateDotFile()

```
template<typename T>
void Graph< T >::generateDotFile (
            const std::string & filename)  [private]
```

Generates a DOT file representing the graph.

Parameters

filename	The name of the DOT file to be generated.

Definition at line 233 of file graph.hpp.

```
00233                                                   {
00234     std::ofstream file(filename);
00235     file « "digraph G {\n";
00236     for (const auto& [node, neighbors] : adjList) {
00237         for (const auto& neighbor : neighbors) {
00238             file « "    \"" « node « "\" -> \"" « neighbor « "\";\n";
00239         }
00240     }
00241     file « "}\n";
00242     file.close();
00243 }
```

Here is the caller graph for this function:

6.2.2.6 hasCycle()

```
template<typename T>
std::optional< std::vector< T > > Graph< T >::hasCycle ()
```

Detects if the graph contains a cycle.

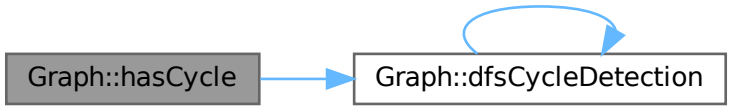
Returns

An optional vector of nodes representing the cycle if one exists, or an empty optional otherwise.

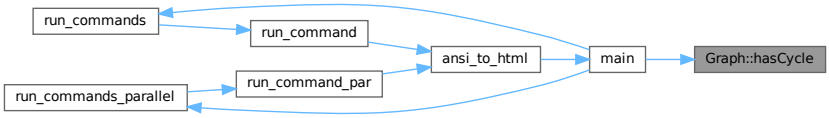
Definition at line 144 of file graph.hpp.

```
00144                                                   {
00145     std::unordered_set<T> visited;
00146     std::unordered_set<T> recStack;
00147     std::vector<T> path;
00148     std::vector<T> cycle;
00149     for (const auto& [node, _] : adjList) {
00150         if (dfsCycleDetection(node, visited, recStack, path, cycle)) {
00151             return cycle;
00152         }
00153     }
00154     return std::nullopt;
00155 }
```

Here is the call graph for this function:



Here is the caller graph for this function:



6.2.2.7 inDegree()

```
template<typename T>
int Graph< T >::inDegree (
            const T & node)
```

Computes the in-degree of a node.

Parameters

node	The node whose in-degree is to be calculated.

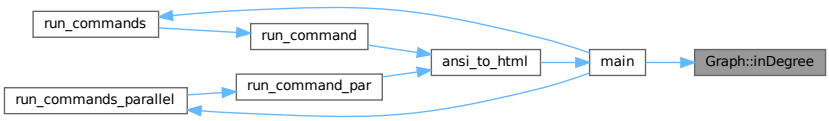
Returns

The in-degree of the node.

Definition at line 209 of file graph.hpp.

```
00209                                       {
00210     int degree = 0;
00211     for (const auto& [key, neighbors] : adjList) {
00212         if (neighbors.find(node) != neighbors.end()) {
00213             degree++;
00214         }
00215     }
00216     return degree;
00217 }
```

Here is the caller graph for this function:



6.2.2.8 outDegree()

```
template<typename T>
int Graph< T >::outDegree (
            const T & node)
```

Computes the out-degree of a node.

Parameters

node	The node whose out-degree is to be calculated.

Returns

The out-degree of the node.

Definition at line 220 of file graph.hpp.

```
00220                                       {
00221     return adjList[node].size();
00222 }
```

6.2.2.9 removeEdge()

```
template<typename T>
void Graph< T >::removeEdge (
            const T & from,
            const T & to)
```

Removes a directed edge between two nodes.

Parameters

from	The source node of the edge.
to	The destination node of the edge.

Definition at line 139 of file graph.hpp.

```
00139                                                   {
00140     adjList[from].erase(to);
00141 }
```

6.2.2.10 removeNode()

```
template<typename T>
void Graph< T >::removeNode (
            const T & node)
```

Removes a node from the graph.

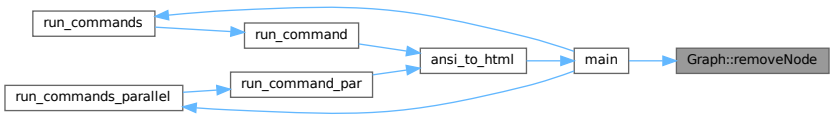
Parameters

node	The node to be removed.

Definition at line 126 of file graph.hpp.

```
00126                                       {
00127     adjList.erase(node);
00128     for (auto& [key, neighbors] : adjList) {
00129         neighbors.erase(node);
00130     }
00131 }
```

Here is the caller graph for this function:



6.2.2.11 topologicalSort()

```
template<typename T>
std::vector< T > Graph< T >::topologicalSort ()
```

Performs a topological sort of the graph.

Returns

A vector of nodes in topologically sorted order.

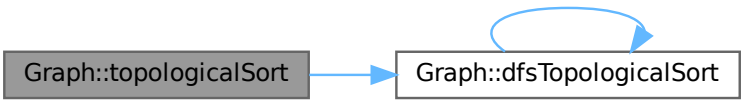
Note

The graph must be a Directed Acyclic Graph (DAG) for this to work correctly.

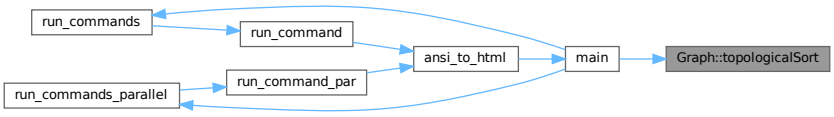
Definition at line 181 of file graph.hpp.

```
00181                                       {
00182     std::stack<T> Stack;
00183     std::unordered_set<T> visited;
00184     for (const auto& [node, _] : adjList) {
00185         if (visited.find(node) == visited.end()) {
00186             dfsTopologicalSort(node, visited, Stack);
00187         }
00188     }
00189     std::vector<T> result;
00190     while (!Stack.empty()) {
00191         result.push_back(Stack.top());
00192         Stack.pop();
00193     }
00194     return result;
00195 }
```

Here is the call graph for this function:



Here is the caller graph for this function:



6.2.2.12 visualize()

```
template<typename T>
void Graph< T >::visualize (
            const std::string & filename,
            const std::string & imgfilename)
```

Visualizes the graph by generating a DOT file and an image.

Parameters

filename	The name of the DOT file to be generated.
imgfilename	The name of the image file to be generated.

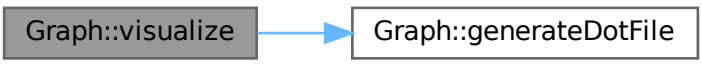
Note

Requires Graphviz to generate the image from the DOT file.

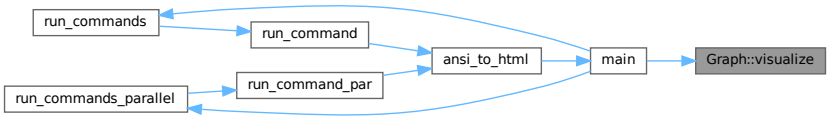
Definition at line 225 of file graph.hpp.

```
00225                                                                    {
00226     generateDotFile(filename);
00227     std::string command = "dot -Tpng " + filename + " -o " + imgfilename + ".png";
00228     system(command.c_str());
00229     std::remove(filename.c_str());
00230 }
```

Here is the call graph for this function:



Here is the caller graph for this function:



6.2.3 Member Data Documentation

6.2.3.1 adjList

```
template<typename T>
std::unordered_map<T, std::unordered_set<T> > Graph< T >::adjList  [private]
```

The adjacency list representation of the graph. Maps each node to a set of its neighboring nodes.

Definition at line 92 of file graph.hpp.

The documentation for this class was generated from the following file:

- graph.hpp

6.3 std::hash< Node > Struct Reference

Public Member Functions

- std::size_t operator() (const Node &node) const

  Computes the hash value for a given `Node`.

6.3.1 Detailed Description

Definition at line 197 of file main.cpp.

6.3.2 Member Function Documentation

6.3.2.1 operator()()

```
std::size_t std::hash< Node >::operator() (
            const Node & node) const  [inline]
```

Computes the hash value for a given `Node`.

Parameters

node	The `Node` object for which the hash value is to be computed.

Returns

The hash value of the `Node`'s `name` attribute.

Definition at line 204 of file main.cpp.

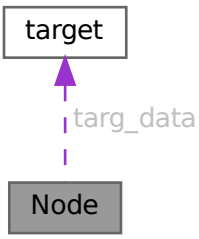
```
00204                                                {
00205             return std::hash<std::string>()(node.name);
00206         }
```

The documentation for this struct was generated from the following file:

- main.cpp

6.4 Node Struct Reference

Represents a node in a dependency graph with associated target data.
Collaboration diagram for Node:



Public Member Functions

- bool operator== (const Node &other) const

  Checks if two nodes are equal based on their names.

- bool operator!= (const Node &other) const

  Checks if two nodes are not equal based on their names.
- bool operator< (const Node &other) const

  Compares two nodes for ordering based on their names.

Public Attributes

- std::string name

  name of the node
- target targ_data

  target data associated with the node

Friends

- std::ostream & operator<< (std::ostream &os, const Node &node)

  Outputs the node's name to an output stream.

6.4.1 Detailed Description

Represents a node in a dependency graph with associated target data.
This struct encapsulates a node's name and its associated target data. It provides comparison operators for equality, inequality, and ordering, as well as a friend function for outputting the node's name to an output stream.
Definition at line 145 of file main.cpp.

6.4.2 Member Function Documentation

6.4.2.1 operator"!=()

```
bool Node::operator!= (
            const Node & other) const  [inline]
```

Checks if two nodes are not equal based on their names.

Parameters

other	The other node to compare with.

Returns

True if the names are not equal, false otherwise.

Definition at line 163 of file main.cpp.

```
00163                                           {
00164         return !(*this == other);
00165     }
```

6.4.2.2 operator<()

```
bool Node::operator< (
            const Node & other) const  [inline]
```

Compares two nodes for ordering based on their names.

Parameters

other	The other node to compare with.

Returns

True if this node's name is lexicographically less than the other's name.

Definition at line 172 of file main.cpp.

```
00172                                           {
00173         return name < other.name;
00174     }
```

6.4.2.3 operator==()

```
bool Node::operator== (
            const Node & other) const  [inline]
```

Checks if two nodes are equal based on their names.

Parameters

other	The other node to compare with.

Returns

True if the names are equal, false otherwise.

Definition at line 154 of file main.cpp.

```
00154                                                {
00155         return name == other.name;
00156     }
```

6.4.3 Friends And Related Symbol Documentation

6.4.3.1 operator<<

```
std::ostream & operator<< (
            std::ostream & os,
            const Node & node)  [friend]
```

Outputs the node's name to an output stream.

Parameters

os	The output stream.
node	The node to output.

Returns

The output stream with the node's name appended.

Definition at line 182 of file main.cpp.

```
00182                                                                {
00183         os << node.name;
00184         return os;
00185     }
```

6.4.4 Member Data Documentation

6.4.4.1 name

```
std::string Node::name
```

name of the node

Definition at line 146 of file main.cpp.

6.4.4.2 targ_data

```
target Node::targ_data
```

target data associated with the node

Definition at line 147 of file main.cpp.

The documentation for this struct was generated from the following file:

- main.cpp

6.5 Parser Class Reference

Class for parsing a build configuration file.
`#include <parser.hpp>`

Public Member Functions

- Parser (const std::string &file)
  Constructs a Parser object with the given file name.
- void printResults () const
  Prints the parsed results (variables, targets, and imports).
- std::tuple< var_table, targ_table, import_table > get_parsed_results ()
  Retrieves the parsed results as a tuple.
- void parse ()
  Parses the file and populates the variables, targets, and imports.
- void set_file_name (std::string &name)
  Sets the name of the file to be parsed.
- std::string get_first_target ()
  Retrieves the name of the first target in the file.

Private Member Functions

- void trim (std::string &s)
  Trims leading and trailing whitespace from a string.
- void parsefile (std::ifstream &file)
  Parses the contents of the file and populates the tables.

Private Attributes

- std::string filename
  The name of the file to be parsed.
- var_table variables
  Table of variables defined in the file.
- targ_table targets
  Table of targets defined in the file.
- import_table imports
  Table of imported files.
- std::string first_target
  The name of the first target in the file.
- std::string currentTarget
  The name of the current target being parsed.
- int line_no = 0
  The current line number being processed.

6.5.1 Detailed Description

Class for parsing a build configuration file.
Definition at line 123 of file parser.hpp.

6.5.2 Constructor & Destructor Documentation

6.5.2.1 Parser()

```
Parser::Parser (
            const std::string & file)  [inline]
```

Constructs a Parser object with the given file name.

Parameters

file	The name of the file to be parsed.

Definition at line 131 of file parser.hpp.

```
00131 : filename(file) {}
```

6.5.3 Member Function Documentation

6.5.3.1 get_first_target()

```
std::string Parser::get_first_target () [inline]
```

Retrieves the name of the first target in the file.

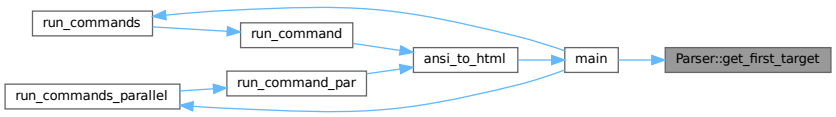
Returns

The name of the first target.

Definition at line 175 of file parser.hpp.

```
00175 { return first_target; }
```

Here is the caller graph for this function:



6.5.3.2 get_parsed_results()

```
std::tuple< var_table, targ_table, import_table > Parser::get_parsed_results () [inline]
```

Retrieves the parsed results as a tuple.

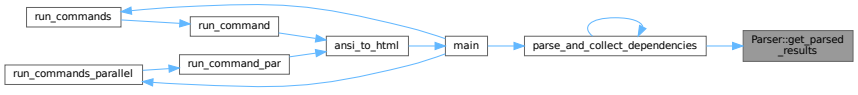
Returns

A tuple containing the variable table, target table, and import table.

Definition at line 143 of file parser.hpp.

```
00143                                                                    {
00144             return std::make_tuple(std::move(variables), std::move(targets), std::move(imports));
00145         }
```

Here is the caller graph for this function:



6.5.3.3 parse()

```
void Parser::parse () [inline]
```

Parses the file and populates the variables, targets, and imports.

Note

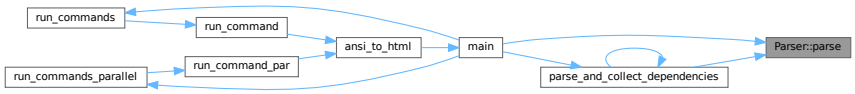
Exits the program if the file cannot be opened.

Definition at line 152 of file parser.hpp.

```
00152                     {
00153             std::ifstream file(filename);
00154             if (!file) {
00155                 std::cerr « "Error opening forgefile. No such file found." « std::endl;
00156                 exit(1);
00157             }
00158             parsefile(file);
00159         }
```

Here is the call graph for this function:

Here is the caller graph for this function:



6.5.3.4 parsefile()

```
void Parser::parsefile (
            std::ifstream & file)  [private]
```

Parses the contents of the file and populates the tables.

Parameters

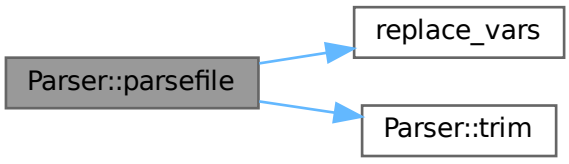
file	The input file stream to be parsed.

Definition at line 35 of file parser.cpp.

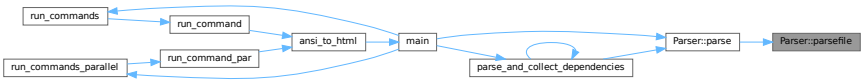
```
00035                                       {
00036     bool is_first_target = true;
00037     std::string line;
00038     bool line_cont = false;
00039     while (std::getline(file, line)) {
00040         line_no++;
00041         trim(line);
00042 
00043         // print the exact characters of the line in ascii
00044         if (line.empty() || line[0] == '#') continue;
00045         replace_vars(line, variables);
00046 
00047         if (line[0] == '\t') {
00048             if (line.length() == 1) continue;
00049             if (!currentTarget.empty()) {
00050                 if (!line.empty() && line.back() == '\\') {
00051                     line.pop_back();
00052                     std::string last_cmd = "";
00053                     if (line_cont) {
00054                         last_cmd = targets[currentTarget].commands.back();
```

```
00055                         targets[currentTarget].commands.pop_back();
00056                     }
00057                     last_cmd += line.substr(1);
00058                     targets[currentTarget].commands.push_back(last_cmd);
00059                     line_cont = true;
00060                     continue;
00061                 }
00062 
00063                 if (line_cont){
00064                     std::string last_cmd = targets[currentTarget].commands.back();
00065                     targets[currentTarget].commands.pop_back();
00066                     last_cmd += line.substr(1);
00067                     targets[currentTarget].commands.push_back(last_cmd);
00068                     line_cont = false;
00069                 } else {
00070                     targets[currentTarget].commands.push_back(line.substr(1));
00071                 }
00072             }
00073         } else if (line.find(":") != std::string::npos &&
00074                     (line.find("=") == std::string::npos || line.find(":") < line.find("="))) {
00075 
00076             size_t colonPos = line.find(":");
00077             currentTarget = line.substr(0, colonPos);
00078             trim(currentTarget);
00079             if(is_first_target) {
00080                 first_target = currentTarget;
00081                 is_first_target = false;
00082             }
00083             std::istringstream depStream(line.substr(colonPos + 1));
00084             std::string dep;
00085             while (depStream » dep) {
00086                 trim(dep);
00087                 targets[currentTarget].dependencies.insert(dep);
00088             }
00089         } else if (line.find("=") != std::string::npos) {
00090             size_t eqPos = line.find("=");
00091             std::string var = line.substr(0, eqPos);
00092             std::string value = line.substr(eqPos + 1);
00093             trim(var);
00094             trim(value);
00095             std::string after_assign = line.substr(eqPos + 1, 7);
00096             trim(after_assign);
00097             if (after_assign == "import") {
00098                 std::string import_str = line.substr(eqPos + 8);
00099                 trim(import_str);
00100                 import_str = import_str.substr(1, import_str.size()- 2);
00101                 imports[var] = import_str;
00102             } else {
00103                 variables[var] = value;
00104             }
00105         } else {
00106             LOG(ERROR, " Error parsing forgefile in line " « line_no « ": " « line);
00107             exit(0);
00108         }
00109     }
00110 }
```

Here is the call graph for this function:



Here is the caller graph for this function:



6.5.3.5 printResults()

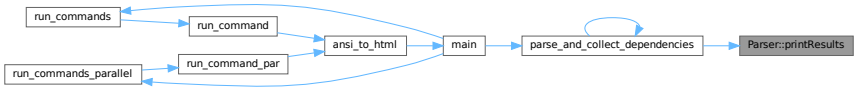
```
void Parser::printResults () const
```

Prints the parsed results (variables, targets, and imports).

Definition at line 112 of file parser.cpp.

```
00112                                {
00113     LOG(INFO, "Variables:\n");
00114     for (const auto &var : variables) {
00115         LOG(INFO, var.first « " = " « var.second « "\n");
00116     }
00117 
00118     LOG(INFO, "\nTargets:\n");
00119     for (const auto &t : targets) {
00120         LOG(INFO, t.first « ": ");
00121         for (const auto &dep : t.second.dependencies) {
00122             LOG(INFO, dep « " ");
00123         }
00124         LOG(INFO, "\nCommands:\n");
00125         for (const auto &cmd : t.second.commands) {
00126             LOG(INFO, cmd « "\n");
00127         }
00128 
00129         LOG(INFO, std::endl);
00130     }
00131 
00132     LOG(INFO, "\nImports:\n");
00133     for (const auto &imp : imports) {
00134         LOG(INFO, imp.first « " = import " « imp.second « "\n");
00135     }
00136 }
```

Here is the caller graph for this function:



6.5.3.6 set_file_name()

```
void Parser::set_file_name (
            std::string & name)  [inline]
```

Sets the name of the file to be parsed.

Parameters

name	The new file name.

Definition at line 166 of file parser.hpp.

```
00166                                          {
00167             filename = name;
00168         }
```

6.5.3.7 trim()

```
void Parser::trim (
            std::string & s)  [private]
```

Trims leading and trailing whitespace from a string.

Parameters

s	The string to be trimmed.

Definition at line 13 of file parser.cpp.

```
00013                           {
00014         if(s[0] == '\t') return;
00015         size_t start = s.find_first_not_of(" \t");
00016         size_t end = s.find_last_not_of(" \t");
00017         s = (start == std::string::npos) ? "" : s.substr(start, end - start + 1);
00018 }
```

Here is the caller graph for this function:

6.5.4 Member Data Documentation

6.5.4.1 currentTarget

`std::string Parser::currentTarget [private]`

The name of the current target being parsed.
Definition at line 184 of file parser.hpp.

6.5.4.2 filename

`std::string Parser::filename [private]`

The name of the file to be parsed.
Definition at line 178 of file parser.hpp.

6.5.4.3 first_target

`std::string Parser::first_target [private]`

The name of the first target in the file.
Definition at line 183 of file parser.hpp.

6.5.4.4 imports

`import_table Parser::imports [private]`

Table of imported files.
Definition at line 181 of file parser.hpp.

6.5.4.5 line_no

`int Parser::line_no = 0 [private]`

The current line number being processed.
Definition at line 185 of file parser.hpp.

6.5.4.6 targets

`targ_table Parser::targets [private]`

Table of targets defined in the file.
Definition at line 180 of file parser.hpp.

6.5.4.7 variables

```
var_table Parser::variables  [private]
```

Table of variables defined in the file.
Definition at line 179 of file parser.hpp.
The documentation for this class was generated from the following files:

- parser.hpp
- parser.cpp

6.6 target Struct Reference

Struct representing a build target with its dependencies and commands.
`#include <parser.hpp>`

Public Attributes

- std::set< std::string > dependencies
- std::vector< std::string > commands

Friends

- std::ostream & operator<< (std::ostream &os, const target &targ)

  Overloads the stream insertion operator to print the target's details.

6.6.1 Detailed Description

Struct representing a build target with its dependencies and commands.
Definition at line 84 of file parser.hpp.

6.6.2 Friends And Related Symbol Documentation

6.6.2.1 operator<<

```
std::ostream & operator<< (
            std::ostream & os,
            const target & targ)  [friend]
```

Overloads the stream insertion operator to print the target's details.

Parameters

os	The output stream.
targ	The target to be printed.

Returns

The output stream with the target's details.

Definition at line 95 of file parser.hpp.

```
00095                                                                    {
00096         os « "Dependencies[";
00097         for (auto dep : targ.dependencies) {
00098             os « dep;
00099             if (dep != *targ.dependencies.rbegin()) {
00100                 os « ", ";
00101             }
00102         }
00103         os « "] Commands[";
00104         for (size_t i = 0; i < targ.commands.size(); ++i) {
00105             os « targ.commands[i];
00106             if (i != targ.commands.size() - 1) {
00107                 os « ", ";
00108             }
```

```
00109         }
00110         os « "]";
00111
00112         return os;
00113     }
```

6.6.3 Member Data Documentation

6.6.3.1 commands

`std::vector<std::string> target::commands`
Definition at line 86 of file parser.hpp.

6.6.3.2 dependencies

`std::set<std::string> target::dependencies`
Definition at line 85 of file parser.hpp.
The documentation for this struct was generated from the following file:

- parser.hpp

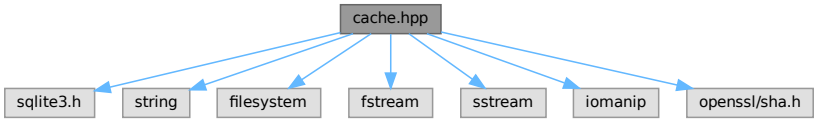
Chapter 7

File Documentation

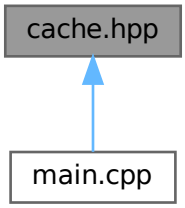
7.1 cache.hpp File Reference

```
#include <sqlite3.h>
#include <string>
#include <filesystem>
#include <fstream>
#include <sstream>
#include <iomanip>
#include <openssl/sha.h>
```

Include dependency graph for cache.hpp:



This graph shows which files directly or indirectly include this file:



Classes

- class Cache

  A class to manage a file cache using an SQLite database.

7.2 cache.hpp

Go to the documentation of this file.

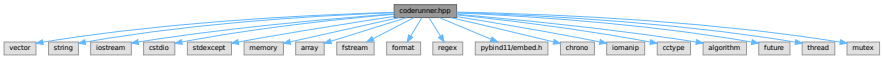
```
00001 #ifndef CACHE_HPP
00002 #define CACHE_HPP
00003
00004 #include <sqlite3.h>
00005 #include <string>
00006 #include <filesystem>
00007 #include <fstream>
00008 #include <sstream>
00009 #include <iomanip>
00010 #include <openssl/sha.h>
00011
00018 class Cache {
00019 private:
00020     sqlite3* db;
00021     std::string db_path;
00022
00030     std::string computeHash(const std::string& file_path) {
00031         std::ifstream file(file_path, std::ios::binary);
00032         if (!file.is_open()) {
00033             throw std::runtime_error("Unable to open file: " + file_path);
00034         }
00035
00036         SHA256_CTX sha256;
00037         SHA256_Init(&sha256);
00038
00039         char buffer[8192];
00040         while (file.read(buffer, sizeof(buffer))) {
00041             SHA256_Update(&sha256, buffer, file.gcount());
00042         }
00043         SHA256_Update(&sha256, buffer, file.gcount());
00044
00045         unsigned char hash[SHA256_DIGEST_LENGTH];
00046         SHA256_Final(hash, &sha256);
00047
00048         std::ostringstream oss;
00049         for (unsigned char c : hash) {
00050             oss << std::hex << std::setw(2) << std::setfill('0') << static_cast<int>(c);
00051         }
00052         return oss.str();
00053     }
00054
00060     void initializeDB() {
00061         const char* create_table_query = R"(
00062             CREATE TABLE IF NOT EXISTS cache (
00063                 file_addr TEXT PRIMARY KEY,
00064                 file_hash TEXT
00065             );
00066         )";
00067
00068         char* err_msg = nullptr;
00069         if (sqlite3_exec(db, create_table_query, nullptr, nullptr, &err_msg) != SQLITE_OK) {
00070             std::string error = "Failed to create table: ";
00071             error += err_msg;
00072             sqlite3_free(err_msg);
00073             throw std::runtime_error(error);
00074         }
00075     }
00076
00077 public:
00085     Cache() {
00086         const char* home = std::getenv("HOME");
00087         if (!home) {
00088             throw std::runtime_error("HOME environment variable not set");
00089         }
00090
00091         db_path = std::string(home) + "/.forgecache";
00092
00093         if (sqlite3_open(db_path.c_str(), &db) != SQLITE_OK) {
00094             throw std::runtime_error("Failed to open SQLite database");
00095         }
00096
00097         initializeDB();
00098     }
00099
00103     ~Cache() {
00104         if (db) {
00105             sqlite3_close(db);
00106         }
00107     }
00108
00115     void add(const std::string& file_addr) {
00116         std::string file_hash = computeHash(file_addr);
00117
```

```
00118         const char* insert_query = R"(
00119             INSERT OR REPLACE INTO cache (file_addr, file_hash)
00120             VALUES (?, ?);
00121         )";
00122 
00123         sqlite3_stmt* stmt;
00124         if (sqlite3_prepare_v2(db, insert_query, -1, &stmt, nullptr) != SQLITE_OK) {
00125             throw std::runtime_error("Failed to prepare insert statement");
00126         }
00127 
00128         sqlite3_bind_text(stmt, 1, file_addr.c_str(), -1, SQLITE_STATIC);
00129         sqlite3_bind_text(stmt, 2, file_hash.c_str(), -1, SQLITE_STATIC);
00130 
00131         if (sqlite3_step(stmt) != SQLITE_DONE) {
00132             sqlite3_finalize(stmt);
00133             throw std::runtime_error("Failed to execute insert statement");
00134         }
00135 
00136         sqlite3_finalize(stmt);
00137     }
00138 
00146     std::string find(const std::string& file_addr) {
00147         const char* select_query = R"(
00148             SELECT file_hash FROM cache WHERE file_addr = ?;
00149         )";
00150 
00151         sqlite3_stmt* stmt;
00152         if (sqlite3_prepare_v2(db, select_query, -1, &stmt, nullptr) != SQLITE_OK) {
00153             throw std::runtime_error("Failed to prepare select statement");
00154         }
00155 
00156         sqlite3_bind_text(stmt, 1, file_addr.c_str(), -1, SQLITE_STATIC);
00157 
00158         std::string file_hash;
00159         if (sqlite3_step(stmt) == SQLITE_ROW) {
00160             file_hash = reinterpret_cast<const char*>(sqlite3_column_text(stmt, 0));
00161         }
00162 
00163         sqlite3_finalize(stmt);
00164         return file_hash;
00165     }
00166 
00173     bool check(const std::string& file_addr) {
00174         std::string file_hash = find(file_addr);
00175         if (file_hash.empty()) {
00176             return false;
00177         }
00178 
00179         std::string current_hash = computeHash(file_addr);
00180         return file_hash == current_hash;
00181     }
00182 };
00183 
00184 #endif // CACHE_HPP
```

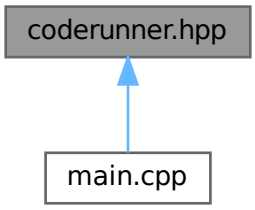
7.3 coderunner.hpp File Reference

```
#include <vector>
#include <string>
#include <iostream>
#include <cstdio>
#include <stdexcept>
#include <memory>
#include <array>
#include <fstream>
#include <format>
#include <regex>
#include <pybind11/embed.h>
#include <chrono>
#include <iomanip>
#include <cctype>
#include <algorithm>
#include <future>
#include <thread>
#include <mutex>
```

Include dependency graph for coderunner.hpp:



This graph shows which files directly or indirectly include this file:



Functions

- int get_exit_code (const std::string &file_path)

  Extracts the exit code from the last line of a file.
- std::string ansi_to_html (const std::string &ansi_text)

  Converts ANSI text to HTML using the Python `ansi2html` library.
- std::string extract_pre_content (const std::string &html)

  Extracts the content inside `<pre>` tags from an HTML string.
- std::string create_html (const std::string &output, std::string &commands)

  Creates an HTML document containing build results and a Gantt chart.
- std::string formatDateTime (const std::chrono::system_clock::time_point &tp)

  Formats a `std::chrono::system_clock::time_point` into a JavaScript `Date` object string.
- std::tuple< std::string, std::string, bool > run_command (const std::vector< std::string > &commands)

  Executes a list of commands sequentially and captures their output.
- std::string run_commands (const std::vector< std::string > &commands)

  Executes a list of commands sequentially and generates an HTML report.
- std::tuple< std::string, std::string, bool > run_command_par (const std::vector< std::string > &commands)

  Executes a list of commands sequentially in parallel and captures their output.
- std::string run_commands_parallel (const std::vector< std::vector< std::vector< std::string > > > &command_batches, size_t num_threads)

  Executes batches of commands in parallel using multiple threads and generates an HTML report.

Variables

- std::mutex py_mutex
- std::mutex stdout_mutex
- html = output
- body
- padding __pad0__
- </style ></head >< body class="body_foreground body_background">< h1 class="forge-box">< a href="https: </h1><h2 style="color:black;"> Build Summary

- Compilation Details
- auto base = std::chrono::system_clock::from_time_t(0)
- int counter = 0

7.3.1 Function Documentation

7.3.1.1 ansi_to_html()

```
std::string ansi_to_html (
            const std::string & ansi_text)
```

Converts ANSI text to HTML using the Python `ansi2html` library.

Parameters

ansi_text	The ANSI-formatted text to be converted.

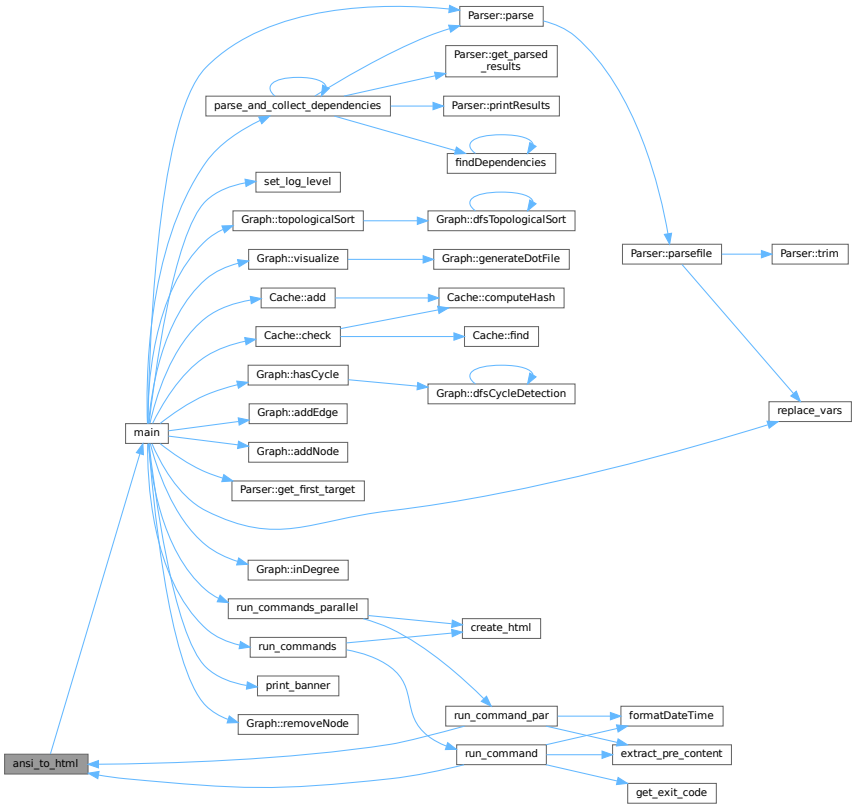
Returns

The converted HTML string.

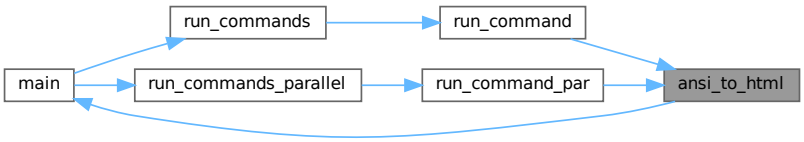
Definition at line 66 of file coderunner.hpp.

```
00066                                                   {
00067     std::lock_guard<std::mutex> lock(py_mutex);
00068     py::scoped_interpreter guard{};
00069 
00070     try {
00071         py::module sys = py::module::import("sys");
00072 
00073         std::string python_code = R"(
00074 from ansi2html import Ansi2HTMLConverter
00075 
00076 def convert_ansi_to_html(ansi_text):
00077     conv = Ansi2HTMLConverter()
00078     html_text = conv.convert(ansi_text)
00079     return html_text
00080 )";
00081 
00082             py::exec(python_code.c_str());
00083 
00084         py::module main = py::module::import("__main__");
00085         py::object convert_ansi_to_html = main.attr("convert_ansi_to_html");
00086 
00087         py::object result = convert_ansi_to_html(ansi_text);
00088 
00089         return result.cast<std::string>();
00090     } catch (const py::error_already_set& e) {
00091         std::cerr « "Error: " « e.what() « std::endl;
00092         return "";
00093     }
00094 }
```

Here is the call graph for this function:



Here is the caller graph for this function:



7.3.1.2 create_html()

```
std::string create_html (
            const std::string & output,
            std::string & commands)
```

Creates an HTML document containing build results and a Gantt chart.

Parameters

output	The HTML content for the build results.
commands	The commands and their execution details for the Gantt chart.

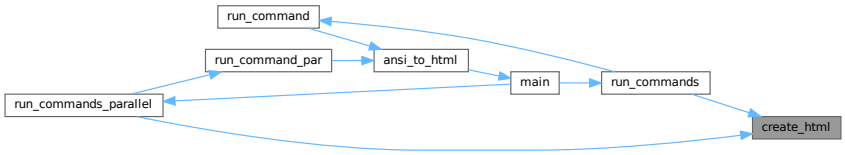
Returns

The complete HTML document as a string.

Definition at line 142 of file coderunner.hpp.

```
00142                                                                 {
00143     std::string html = R"(<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
      "http://www.w3.org/TR/html4/loose.dtd">
00144 <html>
00145 <head>
00146   <meta http-equiv="Content-Type" content="text/html; charset=utf-8">
00147   <title>Forge Build Timeline</title>
00148
00149   <script type="text/javascript" src="https://www.gstatic.com/charts/loader.js"></script>
00150
00151   <style type="text/css">
00152     .ansi2html-content { display: inline; white-space: pre-wrap; word-wrap: break-word; }
00153     .body_foreground { color: #f5f3f3; }
00154     .body_background { background-color: #ffffff; }
00155     .inv_foreground { color: #000000; }
00156     .inv_background { background-color: #faf9f9; }
00157     .ansi1 { font-weight: bold; color: #0b0b0b; }
00158     .ansi2 { font-weight: bold; color: #f7f7f7; }
00159     .ansi32 { color: #00aa00; }
00160     .ansi35 { color: #E850A8; }
00161
00162     .command-box {
00163       border: 2px solid #0332b2;
00164       background-color:  #0332b2;
00165       padding: 5px;
00166       display: inline-block;
00167       border-top-left-radius: 8px;
00168       border-top-right-radius: 8px;
00169       margin-bottom: 0;
00170     }
00171
00172     .forge-box {
00173       border: 2px solid #0e0e0e;
00174       padding: 5px;
00175       color: black;
00176       background-color:  #f6f6f7;
00177       margin-bottom: 0;
00178       display: block;
00179       border-radius: 8px;
00180       text-align: center;
00181       margin: 0 auto;
00182       max-width: 400px;
00183     }
00184
00185     .error-box {
00186       border: 2px solid #0332b2;
00187       padding: 10px;
00188       background-color: #fafafa;
00189       border-bottom-left-radius: 8px;
00190       border-bottom-right-radius: 8px;
00191       border-top-right-radius: 8px;
00192       color: #0b0a0a;
00193       white-space: pre-wrap;
00194       word-wrap: break-word;
00195     }
00196
00197     .gantt-header {
00198       display: flex;
00199       font-weight: bold;
00200       background-color: #600780;
00201       color: white;
00202       padding: 8px 12px;
00203     }
00204
00205     .gantt-header .command-col {
00206       width: 25%;
00207       min-width: 120px;
00208     }
00209
00210     .gantt-header .time-col {
00211       flex: 1;
00212     }
00213
00214     #chart_div {
00215       width: 100%;
00216     }
```

Here is the caller graph for this function:



7.3.1.3 extract_pre_content()

```
std::string extract_pre_content (
            const std::string & html)
```

Extracts the content inside <pre> tags from an HTML string.

Parameters

html	The HTML string to process.

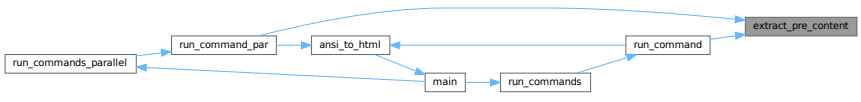
Returns

The extracted content inside <pre> tags, or an empty string if no content is found.

Definition at line 102 of file coderunner.hpp.

```
00102                                                   {
00103     std::vector<std::string> lines;
00104     std::istringstream stream(html);
00105 
00106     std::string output = "";
00107 
00108     bool pre_start = false, pre_end = false;
00109 
00110     int cnt = 0;
00111 
00112     std::string line;
00113     while (std::getline(stream, line)) {
00114         if (line.find("pre ") != std::string::npos) {
00115             pre_start = true;
00116             output += line + "\n";
00117         }
00118         else if (line.find("</pre>") != std::string::npos) {
00119             pre_end = true;
00120             output += line;
00121         }
00122         else if (pre_start && !pre_end) {
00123             if (!std::all_of(line.begin(), line.end(), [](unsigned char ch) { return std::isspace(ch);
      }))
00124                 cnt++;
00125             output += line + "\n";
00126         }
00127     }
00128 
00129     if (cnt == 0) {
00130         return "";
00131     }
00132     return output;
00133 }
```

Here is the caller graph for this function:



7.3.1.4 formatDateTime()

```
std::string formatDateTime (
            const std::chrono::system_clock::time_point & tp)
```

Formats a `std::chrono::system_clock::time_point` into a JavaScript `Date` object string.

Parameters

tp	The time point to format.

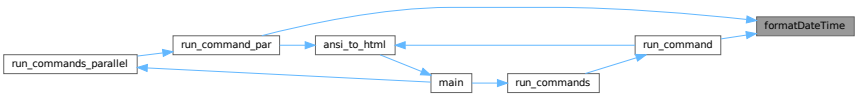
Returns

A string representing the JavaScript `Date` object.

Definition at line 343 of file coderunner.hpp.

```
00343                                                                    {
00344     using namespace std::chrono;
00345
00346     auto duration = tp - base;
00347     auto millis = duration_cast<milliseconds>(duration).count();
00348
00349     auto hours = millis / (1000 * 60 * 60);
00350     millis %= (1000 * 60 * 60);
00351     auto minutes = millis / (1000 * 60);
00352     millis %= (1000 * 60);
00353     auto seconds = millis / 1000;
00354     auto milliseconds_part = millis % 1000;
00355
00356     std::ostringstream oss;
00357     oss « "new Date(1970, 0, 1, "
00358         « hours « ", "
00359         « minutes « ", "
00360         « seconds « ", "
00361         « milliseconds_part « ")";
00362
00363     return oss.str();
00364 }
```

Here is the caller graph for this function:



7.3.1.5 get_exit_code()

```
int get_exit_code (
            const std::string & file_path)
```

Extracts the exit code from the last line of a file.

Parameters

file_path	The path to the file containing the exit code.

Returns

The extracted exit code as an integer.

Exceptions

std::runtime_error	If the file cannot be opened or the exit code is not found.

Definition at line 36 of file coderunner.hpp.

```
00036                                                {
00037     std::ifstream file(file_path);
00038     if (!file.is_open()) {
00039         throw std::runtime_error("Could not open file");
00040     }
00041
00042     std::string line;
00043     std::string last_line;
00044     while (std::getline(file, line)) {
00045         last_line = line;
00046     }
00047
00048     file.close();
00049
00050     std::regex exit_code_regex(R"(\[COMMAND_EXIT_CODE="(\d+)\"])");
00051     std::smatch match;
00052     if (std::regex_search(last_line, match, exit_code_regex) && match.size() > 1) {
00053         return std::stoi(match.str(1));
00054     } else {
00055         std::cout « file_path « std::endl;
00056         throw std::runtime_error("Exit code not found in the last line");
00057     }
00058 }
```

Here is the caller graph for this function:

7.3.1.6 run_command()

```
std::tuple< std::string, std::string, bool > run_command (
            const std::vector< std::string > & commands)
```

Executes a list of commands sequentially and captures their output.

Parameters

commands	A vector of commands to execute.

Returns

A tuple containing:

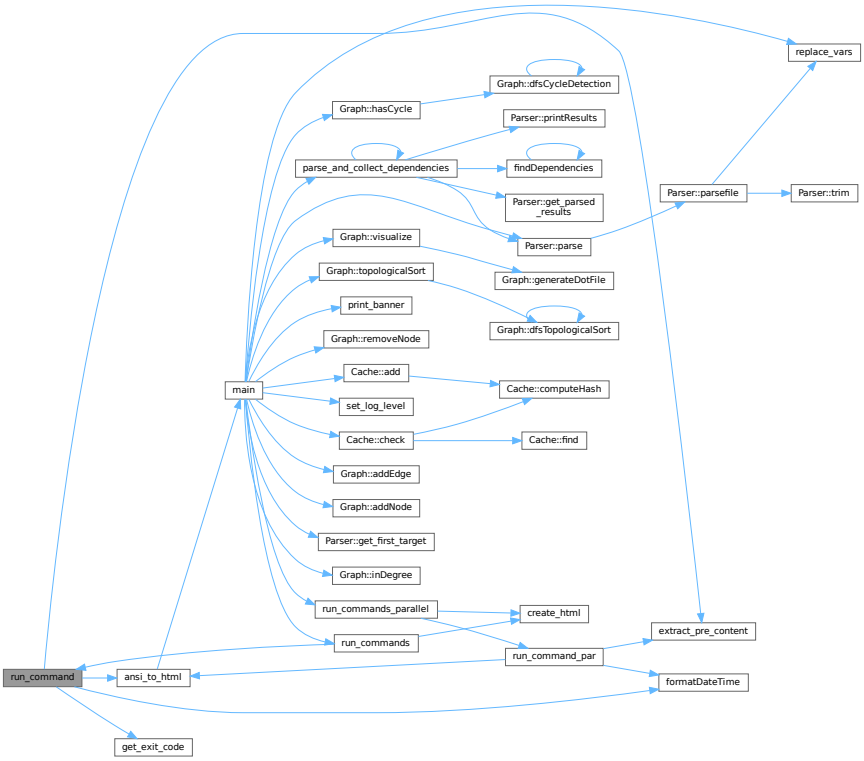
- The HTML-formatted output of the commands.
- The timeline data for the Gantt chart.
- A boolean indicating if any command failed.

Definition at line 378 of file coderunner.hpp.

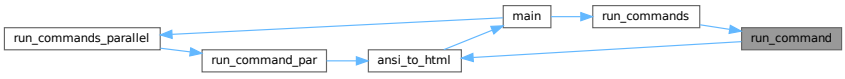
```
00378                                                                                {
00379
00380     timer::time_point<std::chrono::system_clock> start, end;
00381
00382     std::string time;
00383     std::array<char, 128> buffer;
00384     std::string final_result;
00385     bool err = false;
00386     int index = 0;
00387     for (const auto& command : commands) {
00388         counter++;
00389         std::string result;
00390         std::string full_command = "script -q -c \"" + command + "\"" + " 2>&1";
00391
00392         std::cout « command « std::endl;
00393
```

```
00394         start = timer::system_clock::now();
00395         std::unique_ptr<FILE, decltype(&pclose)> pipe(popen(full_command.c_str(), "r"), pclose);
00396         if (!pipe) {
00397             throw std::runtime_error("popen() failed!");
00398         }
00399 
00400         while (fgets(buffer.data(), buffer.size(), pipe.get()) != nullptr) {
00401             result += buffer.data();
00402         }
00403 
00404         std::cout « result « std::endl;
00405 
00406         std::string html_output = ansi_to_html(result);
00407         std::string pre_content = extract_pre_content(html_output);
00408 
00409         std::string start_tags = std::format(R"(<div class="command-box">
00410             <span class="ansi2" id = "cmd{}"> {} </span>
00411             </div>
00412             <div class="error-box">)", counter, command);
00413 
00414         if (pre_content.length() > 0)
00415             pre_content = start_tags + pre_content + "</div><br><br>";
00416 
00417         final_result += pre_content;
00418         bool is_err = false, is_warn = false;
00419         is_err = get_exit_code("typescript") != 0;
00420         if (is_err) err = true;
00421 
00422         if (result.find("warning") != std::string::npos || result.find("Warning") != std::string::npos
      || result.find("WARNING") != std::string::npos) {
00423             is_warn = true;
00424         }
00425 
00426         end = timer::system_clock::now();
00427 
00428         std::string color = is_err ? "red" : (is_warn ? "yellow" : "green");
00429         std::ostringstream ostr;
00430         ostr « "[" « "\'" « counter « "\' , " «  "\'" « command «  "\' , " « "\'" « color « "\' , " «
      formatDateTime(start) « " , " « formatDateTime(end) « ", null , " « (is_err ? 0 : 100) « " , null ],";
00431         time += ostr.str();
00432 
00433         if(is_err) break;
00434 
00435         std::remove("typescript");
00436     }
00437 
00438     std::remove("typescript");
00439 
00440     return {final_result, time, err};
00441 }
```

Here is the call graph for this function:



Here is the caller graph for this function:



7.3.1.7 run_command_par()

```
std::tuple< std::string, std::string, bool > run_command_par (
            const std::vector< std::string > & commands)
```

Executes a list of commands sequentially in parallel and captures their output.

Parameters

commands	A vector of commands to execute.

Returns

A tuple containing:

- The HTML-formatted output of the commands.
- The timeline data for the Gantt chart.
- A boolean indicating if any command failed.

Definition at line 485 of file coderunner.hpp.

```
00485                                                                                              {
00486 
00487     timer::time_point<std::chrono::system_clock> start, end;
00488 
00489     std::string time;
00490     std::array<char, 128> buffer;
00491     std::string final_result;
00492     bool err = false;
00493     int index = 0;
00494     for (const auto& command : commands) {
00495         std::string result;
00496         std::string full_command = command +" 2>&1";
00497 
00498         start = timer::system_clock::now();
00499         std::unique_ptr<FILE, decltype(&pclose)> pipe(popen(full_command.c_str(), "r"), pclose);
00500         if (!pipe) {
00501             throw std::runtime_error("popen() failed!");
00502         }
00503 
00504         while (fgets(buffer.data(), buffer.size(), pipe.get()) != nullptr) {
00505             result += buffer.data();
00506         }
00507 
00508         {
00509             std::lock_guard<std::mutex> lock(stdout_mutex);
00510             std::cout « command « std::endl;
00511             std::cout « result « std::endl;
00512         }
00513 
00514         std::string html_output = ansi_to_html(result);
00515         std::string pre_content = extract_pre_content(html_output);
00516         bool is_err = false, is_warn = false;
00517 
00518         auto pip = pipe.release();
00519         is_err = pclose(pip) != 0;
00520 
00521         if (is_err) err = true;
00522 
00523         if (result.find("warning") != std::string::npos || result.find("Warning") != std::string::npos
      || result.find("WARNING") != std::string::npos) {
00524             is_warn = true;
00525         }
00526 
00527         end = timer::system_clock::now();
00528 
00529         std::string color = is_err ? "red" : (is_warn ? "yellow" : "green");
00530         std::ostringstream ostr;
00531         {
00532             std::lock_guard<std::mutex> lock(stdout_mutex);
00533             counter++;
00534             ostr « "[" « "\'" « counter « "\' , " «  "\'" « command «  "\' , " « "\'" « color « "\' ,
      " « formatDateTime(start) « " , " « formatDateTime(end) « ", null , " « (is_err ? 0 : 100) « " , null
      ],";
00535             time += ostr.str();
00536             std::string start_tags = std::format(R"(<div class="command-box">
00537             <span class="ansi2" id = "cmd{}"> {} </span>
00538             </div>
00539             <div class="error-box">)", counter, command);
00540 
00541             if (pre_content.length() > 0)
00542                 pre_content = start_tags + pre_content + "</div><br><br>";
00543 
00544             final_result += pre_content;
00545         }
00546 
00547         if(is_err) break;
00548     }
00549 
00550     return {final_result, time, err};
00551 }
```

Here is the call graph for this function:

Here is the caller graph for this function:

7.3.1.8 run_commands()

```
std::string run_commands (
            const std::vector< std::string > & commands)
```

Executes a list of commands sequentially and generates an HTML report.

Parameters

commands	A vector of commands to execute.

Returns

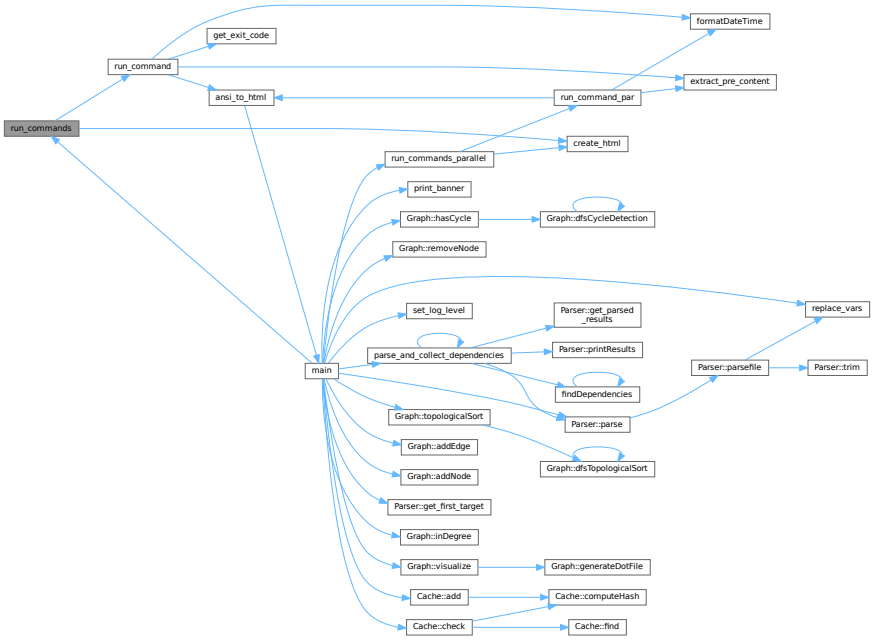
The HTML report as a string.

Definition at line 449 of file coderunner.hpp.

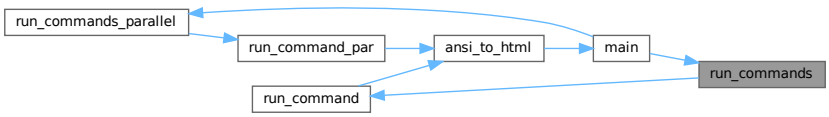
```
00449                                                                {
00450     std::string time;
00451     std::string final_result;
00452     bool is_err = false;
00453
00454     using timer = std::chrono::system_clock;
00455     timer::time_point start, end;
00456
00457     start = timer::now();
```

```
00458     for (const auto& command : commands) {
00459         auto [result, command_time, err] = run_command({command});
00460         final_result += result;
00461         time += command_time;
00462         if (err) { is_err = true; break; }
00463     }
00464     end = timer::now();
00465     auto elapsed_seconds = std::chrono::duration<double>(end - start).count();
00466 
00467     if (!is_err){
00468         std::cout << "\033[34m" << "Compilation process completed in: " << elapsed_seconds << " seconds" <<
      "\033[0m" << std::endl;
00469     } else {
00470         std::cout << "\033[31m" << "Compilation process completed in: " << elapsed_seconds << " seconds" <<
      "\033[0m" << std::endl;
00471     }
00472 
00473     return create_html(final_result, time);
00474 }
```

Here is the call graph for this function:



Here is the caller graph for this function:



7.3.1.9 run_commands_parallel()

```
std::string run_commands_parallel (
            const std::vector< std::vector< std::vector< std::string > > > & command_↩
batches,
            size_t num_threads)
```

Executes batches of commands in parallel using multiple threads and generates an HTML report.

Parameters

command_batches	A vector of command batches, where each batch is a vector of command groups.
num_threads	The maximum number of threads to use for parallel execution.

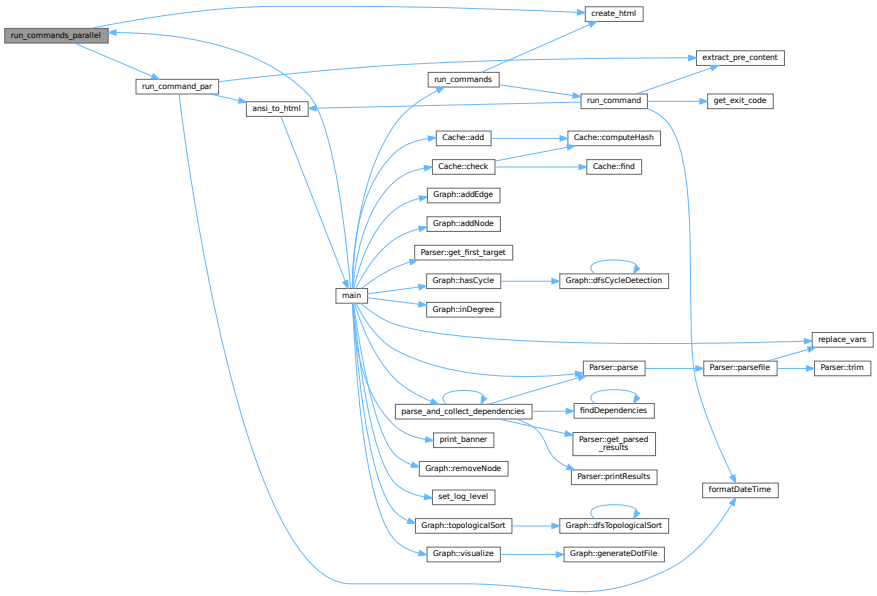
Returns

The HTML report as a string.

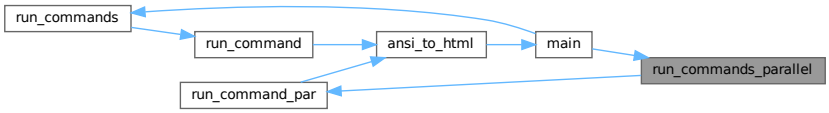
Definition at line 560 of file coderunner.hpp.

```
00560
    {
00561     std::string final_result, total_time;
00562     std::mutex result_mutex;
00563     std::vector<std::thread> threads;
00564     std::atomic<bool> is_err(false);
00565
00566     using clock = std::chrono::system_clock;
00567     auto start = clock::now();
00568
00569     for (const auto& batch : command_batches) {
00570         std::vector<std::future<void» futures;
00571         is_err.store(false);
00572
00573         for (const auto& command : batch) {
00574             futures.emplace_back(std::async(std::launch::async, [&command, &final_result, &total_time,
    &result_mutex, &is_err]() {
00575                 if (is_err.load()) {
00576                     return;
00577                 }
00578
00579                 auto [result, command_time, err] = run_command_par(command);
00580
00581                 {
00582                     std::lock_guard<std::mutex> lock(result_mutex);
00583                     final_result += result;
00584                     total_time += command_time;
00585                 }
00586
00587                 if (err) {
00588                     is_err.store(true);
00589                     return;
00590                 }
00591             }));
00592
00593             if (futures.size() >= num_threads) {
00594                 for (auto& future : futures) {
00595                     future.get();
00596                 }
00597                 futures.clear();
00598             }
00599         }
00600
00601         for (auto& future : futures) {
00602             future.get();
00603         }
00604
00605         if (is_err.load()) {
00606             break;
00607         }
00608     }
00609
00610     auto end = clock::now();
00611     double elapsed = std::chrono::duration<double>(end - start).count();
00612
00613     std::cout « (is_err.load() ? "\033[31m" : "\033[34m")
00614             « "Compilation process completed in: " « elapsed « " seconds"
00615             « "\033[0m" « std::endl;
00616
00617     return create_html(final_result, total_time);
00618 }
```

Here is the call graph for this function:



Here is the caller graph for this function:



7.3.2 Variable Documentation

7.3.2.1 __pad0__

`padding __pad0__`
Definition at line 220 of file coderunner.hpp.

7.3.2.2 base

`auto base = std::chrono::system_clock::from_time_t(0)`
Definition at line 335 of file coderunner.hpp.

7.3.2.3 body

`body`
Initial value:

```
{
      margin: 0
```

Definition at line 218 of file coderunner.hpp.

7.3.2.4 counter

`int counter = 0`
Definition at line 367 of file coderunner.hpp.

7.3.2.5 Details

```
Compilation Details
```

Definition at line 240 of file coderunner.hpp.

7.3.2.6 html

```
return html = output
```

Definition at line 218 of file coderunner.hpp.

7.3.2.7 py_mutex

```
std::mutex py_mutex
```

Definition at line 27 of file coderunner.hpp.

7.3.2.8 stdout_mutex

```
std::mutex stdout_mutex
```

Definition at line 27 of file coderunner.hpp.

7.3.2.9 Summary

```
</style></head><body class="body_foreground body_background"><h1 class="forge-box"><a href="https↩
:  </h1><h2 style="color:  black;"> Build Summary
```

Definition at line 231 of file coderunner.hpp.

7.4 coderunner.hpp

Go to the documentation of this file.

```
00001 #ifndef CODERUNNER_HPP
00002 #define CODERUNNER_HPP
00003
00004 #include <vector>
00005 #include <string>
00006 #include <iostream>
00007 #include <cstdio>
00008 #include <stdexcept>
00009 #include <memory>
00010 #include <array>
00011 #include <fstream>
00012 #include <format>
00013 #include <regex>
00014 #include <pybind11/embed.h>
00015 #include <chrono>
00016 #include <iomanip>
00017 #include <cctype>
00018 #include <algorithm>
00019 #include <future>
00020 #include <thread>
00021 #include <mutex>
00022
00023 namespace timer = std::chrono;
00024
00025 namespace py = pybind11;
00026
00027 std::mutex py_mutex, stdout_mutex;
00028
00036 int get_exit_code(const std::string& file_path) {
00037     std::ifstream file(file_path);
00038     if (!file.is_open()) {
00039         throw std::runtime_error("Could not open file");
00040     }
00041
00042     std::string line;
00043     std::string last_line;
00044     while (std::getline(file, line)) {
00045         last_line = line;
00046     }
00047
00048     file.close();
00049
00050     std::regex exit_code_regex(R"(\[COMMAND_EXIT_CODE="(\d+)\"])");
00051     std::smatch match;
```

```
00052     if (std::regex_search(last_line, match, exit_code_regex) && match.size() > 1) {
00053         return std::stoi(match.str(1));
00054     } else {
00055         std::cout « file_path « std::endl;
00056         throw std::runtime_error("Exit code not found in the last line");
00057     }
00058 }
00059
00066 std::string ansi_to_html(const std::string& ansi_text) {
00067     std::lock_guard<std::mutex> lock(py_mutex);
00068     py::scoped_interpreter guard{};
00069
00070     try {
00071         py::module sys = py::module::import("sys");
00072
00073         std::string python_code = R"(
00074 from ansi2html import Ansi2HTMLConverter
00075
00076 def convert_ansi_to_html(ansi_text):
00077     conv = Ansi2HTMLConverter()
00078     html_text = conv.convert(ansi_text)
00079     return html_text
00080 )";
00081
00082             py::exec(python_code.c_str());
00083
00084         py::module main = py::module::import("__main__");
00085         py::object convert_ansi_to_html = main.attr("convert_ansi_to_html");
00086
00087         py::object result = convert_ansi_to_html(ansi_text);
00088
00089         return result.cast<std::string>();
00090     } catch (const py::error_already_set& e) {
00091         std::cerr « "Error: " « e.what() « std::endl;
00092         return "";
00093     }
00094 }
00095
00102 std::string extract_pre_content(const std::string& html) {
00103     std::vector<std::string> lines;
00104     std::istringstream stream(html);
00105
00106     std::string output = "";
00107
00108     bool pre_start = false, pre_end = false;
00109
00110     int cnt = 0;
00111
00112     std::string line;
00113     while (std::getline(stream, line)) {
00114         if (line.find("pre ") != std::string::npos) {
00115             pre_start = true;
00116             output += line + "\n";
00117         }
00118         else if (line.find("</pre>") != std::string::npos) {
00119             pre_end = true;
00120             output += line;
00121         }
00122         else if (pre_start && !pre_end) {
00123             if (!std::all_of(line.begin(), line.end(), [](unsigned char ch) { return std::isspace(ch);
      }))
00124                 cnt++;
00125             output += line + "\n";
00126         }
00127     }
00128
00129     if (cnt == 0) {
00130         return "";
00131     }
00132     return output;
00133 }
00134
00142 std::string create_html(const std::string& output, std::string& commands) {
00143     std::string html = R"(<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
      "http://www.w3.org/TR/html4/loose.dtd">
00144 <html>
00145 <head>
00146   <meta http-equiv="Content-Type" content="text/html; charset=utf-8">
00147   <title>Forge Build Timeline</title>
00148
00149   <script type="text/javascript" src="https://www.gstatic.com/charts/loader.js"></script>
00150
00151   <style type="text/css">
00152     .ansi2html-content { display: inline; white-space: pre-wrap; word-wrap: break-word; }
00153     .body_foreground { color: #f5f3f3; }
00154     .body_background { background-color: #ffffff; }
00155     .inv_foreground { color: #000000; }
```

```
00156     .inv_background { background-color: #faf9f9; }
00157     .ansi1 { font-weight: bold; color: #0b0b0b; }
00158     .ansi2 { font-weight: bold; color: #f7f7f7; }
00159     .ansi32 { color: #00aa00; }
00160     .ansi35 { color: #E850A8; }
00161
00162     .command-box {
00163       border: 2px solid #0332b2;
00164       background-color:  #0332b2;
00165       padding: 5px;
00166       display: inline-block;
00167       border-top-left-radius: 8px;
00168       border-top-right-radius: 8px;
00169       margin-bottom: 0;
00170     }
00171
00172     .forge-box {
00173       border: 2px solid #0e0e0e;
00174       padding: 5px;
00175       color: black;
00176       background-color:  #f6f6f7;
00177       margin-bottom: 0;
00178       display: block;
00179       border-radius: 8px;
00180       text-align: center;
00181       margin: 0 auto;
00182       max-width: 400px;
00183     }
00184
00185     .error-box {
00186       border: 2px solid #0332b2;
00187       padding: 10px;
00188       background-color: #fafafa;
00189       border-bottom-left-radius: 8px;
00190       border-bottom-right-radius: 8px;
00191       border-top-right-radius: 8px;
00192       color: #0b0a0a;
00193       white-space: pre-wrap;
00194       word-wrap: break-word;
00195     }
00196
00197     .gantt-header {
00198       display: flex;
00199       font-weight: bold;
00200       background-color: #600780;
00201       color: white;
00202       padding: 8px 12px;
00203     }
00204
00205     .gantt-header .command-col {
00206       width: 25%;
00207       min-width: 120px;
00208     }
00209
00210     .gantt-header .time-col {
00211       flex: 1;
00212     }
00213
00214     #chart_div {
00215       width: 100%;
00216     }
00217
00218     html, body {
00219       margin: 0;
00220       padding: 1rem;
00221     }
00222   </style>
00223 </head>
00224
00225 <body class="body_foreground body_background">
00226
00227   <h1 class="forge-box">
00228     <a href="https://github.com/Kronos-192081/forge">forge</a> build results
00229   </h1>
00230
00231   <h2 style="color: black;">Build Summary:</h2>
00232   <div class="gantt-header">
00233     <div class="command-col">Command</div>
00234     <div class="time-col">Timeline</div>
00235   </div>
00236   <div id="chart_div"></div>
00237
00238   <br>
00239
00240   <h2 style="color: black;">Compilation Details:</h2>)";
00241
00242 html += output;
```

```
00243
00244 html += R"(<script>
00245     google.charts.load('current', { packages: ['gantt'] });
00246     google.charts.setOnLoadCallback(drawChart);
00247
00248     function drawChart() {
00249       const container = document.getElementById('chart_div');
00250       const width = container.getBoundingClientRect().width;
00251       const rowHeight = 30;
00252
00253       const data = new google.visualization.DataTable();
00254       data.addColumn('string', 'Task ID');
00255       data.addColumn('string', 'Command');
00256       data.addColumn('string', 'Resource');
00257       data.addColumn('date', 'Start');
00258       data.addColumn('date', 'End');
00259       data.addColumn('number', 'Duration');
00260       data.addColumn('number', 'Percent Complete');
00261       data.addColumn('string', 'Dependencies');
00262
00263       const rows = [)";
00264
00265 html += commands;
00266
00267 html += R"(];
00268
00269       data.addRows(rows);
00270
00271       const colorMap = {
00272         'green': '#4CAF50',
00273         'red': '#F44336',
00274         'yellow': '#FFC107'
00275       };
00276
00277     const palt = [];
00278     const colorSet = new Set();
00279     for (const row of rows) {
00280       const color = colorMap[row[2]];
00281       if (!colorSet.has(color)) {
00282         obj = {
00283         color: color,
00284         dark: color,
00285         light: color
00286         };
00287         palt.push(obj);
00288         colorSet.add(color);
00289       }
00290     }
00291
00292       console.log(palt);
00293
00294       const options = {
00295         height: rows.length * rowHeight + 50,
00296         width: width,
00297         gantt: {
00298           trackHeight: rowHeight,
00299           palette: palt,
00300         }
00301       };
00302
00303       const chart = new google.visualization.Gantt(container);
00304       chart.draw(data, options);
00305
00306       // Clickable bars
00307       const commandLinks = {};
00308         rows.forEach((row, index) => {
00309         const command = row[1];
00310         commandLinks[command] = `#cmd${index + 1}`;
00311      });
00312
00313       google.visualization.events.addListener(chart, 'select', function () {
00314         const selection = chart.getSelection();
00315         if (selection.length > 0) {
00316           const row = selection[0].row;
00317           const command = data.getValue(row, 1);
00318           const link = commandLinks[command];
00319           if (link) {
00320             location.href = link;
00321           }
00322         }
00323       });
00324     }
00325
00326     window.addEventListener('resize', drawChart);
00327   </script>
00328 </body>
00329 </html>)";
```

```
00330
00331     return html;
00332 }
00333
00334
00335 auto base = std::chrono::system_clock::from_time_t(0);
00336
00343 std::string formatDateTime(const std::chrono::system_clock::time_point& tp) {
00344     using namespace std::chrono;
00345
00346     auto duration = tp - base;
00347     auto millis = duration_cast<milliseconds>(duration).count();
00348
00349     auto hours = millis / (1000 * 60 * 60);
00350     millis %= (1000 * 60 * 60);
00351     auto minutes = millis / (1000 * 60);
00352     millis %= (1000 * 60);
00353     auto seconds = millis / 1000;
00354     auto milliseconds_part = millis % 1000;
00355
00356     std::ostringstream oss;
00357     oss « "new Date(1970, 0, 1, "
00358         « hours « ", "
00359         « minutes « ", "
00360         « seconds « ", "
00361         « milliseconds_part « ")";
00362
00363     return oss.str();
00364 }
00365
00366
00367 int counter = 0;
00368
00378 std::tuple<std::string, std::string, bool> run_command(const std::vector<std::string>& commands) {
00379
00380     timer::time_point<std::chrono::system_clock> start, end;
00381
00382     std::string time;
00383     std::array<char, 128> buffer;
00384     std::string final_result;
00385     bool err = false;
00386     int index = 0;
00387     for (const auto& command : commands) {
00388         counter++;
00389         std::string result;
00390         std::string full_command = "script -q -c \"" + command + "\"" + " 2>&1";
00391
00392         std::cout « command « std::endl;
00393
00394         start = timer::system_clock::now();
00395         std::unique_ptr<FILE, decltype(&pclose)> pipe(popen(full_command.c_str(), "r"), pclose);
00396         if (!pipe) {
00397             throw std::runtime_error("popen() failed!");
00398         }
00399
00400         while (fgets(buffer.data(), buffer.size(), pipe.get()) != nullptr) {
00401             result += buffer.data();
00402         }
00403
00404         std::cout « result « std::endl;
00405
00406         std::string html_output = ansi_to_html(result);
00407         std::string pre_content = extract_pre_content(html_output);
00408
00409         std::string start_tags = std::format(R"(<div class="command-box">
00410             <span class="ansi2" id = "cmd{}"> {} </span>
00411             </div>
00412             <div class="error-box">)", counter, command);
00413
00414         if (pre_content.length() > 0)
00415             pre_content = start_tags + pre_content + "</div><br><br>";
00416
00417         final_result += pre_content;
00418         bool is_err = false, is_warn = false;
00419         is_err = get_exit_code("typescript") != 0;
00420         if (is_err) err = true;
00421
00422         if (result.find("warning") != std::string::npos || result.find("Warning") != std::string::npos
      || result.find("WARNING") != std::string::npos) {
00423             is_warn = true;
00424         }
00425
00426         end = timer::system_clock::now();
00427
00428         std::string color = is_err ? "red" : (is_warn ? "yellow" : "green");
00429         std::ostringstream ostr;
00430         ostr « "[" « "\'" « counter « "\' , " «  "\'" « command «  "\' , " « "\'" « color « "\' , " «
```

```
    formatDateTime(start) « " , " « formatDateTime(end) « ", null , " « (is_err ? 0 : 100) « " , null ],";
00431         time += ostr.str();
00432
00433         if(is_err) break;
00434
00435         std::remove("typescript");
00436     }
00437
00438     std::remove("typescript");
00439
00440     return {final_result, time, err};
00441 }
00442
00449 std::string run_commands(const std::vector<std::string>& commands){
00450     std::string time;
00451     std::string final_result;
00452     bool is_err = false;
00453
00454     using timer = std::chrono::system_clock;
00455     timer::time_point start, end;
00456
00457     start = timer::now();
00458     for (const auto& command : commands) {
00459         auto [result, command_time, err] = run_command({command});
00460         final_result += result;
00461         time += command_time;
00462         if (err) { is_err = true; break; }
00463     }
00464     end = timer::now();
00465     auto elapsed_seconds = std::chrono::duration<double>(end - start).count();
00466
00467     if (!is_err){
00468         std::cout « "\033[34m" « "Compilation process completed in: " « elapsed_seconds « " seconds" «
    "\033[0m" « std::endl;
00469     } else {
00470         std::cout « "\033[31m" « "Compilation process completed in: " « elapsed_seconds « " seconds" «
    "\033[0m" « std::endl;
00471     }
00472
00473     return create_html(final_result, time);
00474 }
00475
00485 std::tuple<std::string, std::string, bool> run_command_par(const std::vector<std::string>& commands) {
00486
00487     timer::time_point<std::chrono::system_clock> start, end;
00488
00489     std::string time;
00490     std::array<char, 128> buffer;
00491     std::string final_result;
00492     bool err = false;
00493     int index = 0;
00494     for (const auto& command : commands) {
00495         std::string result;
00496         std::string full_command = command +" 2>&1";
00497
00498         start = timer::system_clock::now();
00499         std::unique_ptr<FILE, decltype(&pclose)> pipe(popen(full_command.c_str(), "r"), pclose);
00500         if (!pipe) {
00501             throw std::runtime_error("popen() failed!");
00502         }
00503
00504         while (fgets(buffer.data(), buffer.size(), pipe.get()) != nullptr) {
00505             result += buffer.data();
00506         }
00507
00508         {
00509             std::lock_guard<std::mutex> lock(stdout_mutex);
00510             std::cout « command « std::endl;
00511             std::cout « result « std::endl;
00512         }
00513
00514         std::string html_output = ansi_to_html(result);
00515         std::string pre_content = extract_pre_content(html_output);
00516         bool is_err = false, is_warn = false;
00517
00518         auto pip = pipe.release();
00519         is_err = pclose(pip) != 0;
00520
00521         if (is_err) err = true;
00522
00523         if (result.find("warning") != std::string::npos || result.find("Warning") != std::string::npos
    || result.find("WARNING") != std::string::npos) {
00524             is_warn = true;
00525         }
00526
00527         end = timer::system_clock::now();
00528
```

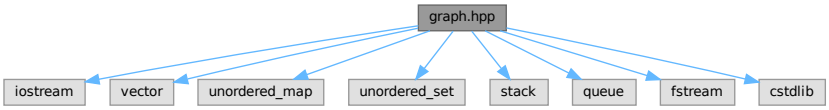
```
00529         std::string color = is_err ? "red" : (is_warn ? "yellow" : "green");
00530         std::ostringstream ostr;
00531         {
00532             std::lock_guard<std::mutex> lock(stdout_mutex);
00533             counter++;
00534             ostr « "[" « "\'" « counter « "\' , " «  "\'" « command «  "\' , " « "\'" « color « "\' ,
      " « formatDateTime(start) « " , " « formatDateTime(end) « ", null , " « (is_err ? 0 : 100) « " , null
      ],";
00535             time += ostr.str();
00536             std::string start_tags = std::format(R"(<div class="command-box">
00537             <span class="ansi2" id = "cmd{}"> {} </span>
00538             </div>
00539             <div class="error-box">)", counter, command);
00540 
00541             if (pre_content.length() > 0)
00542                 pre_content = start_tags + pre_content + "</div><br><br>";
00543 
00544             final_result += pre_content;
00545         }
00546 
00547         if(is_err) break;
00548     }
00549 
00550     return {final_result, time, err};
00551 }
00552 
00560 std::string run_commands_parallel(const std::vector<std::vector<std::vector<std::string»>&
      command_batches, size_t num_threads) {
00561     std::string final_result, total_time;
00562     std::mutex result_mutex;
00563     std::vector<std::thread> threads;
00564     std::atomic<bool> is_err(false);
00565 
00566     using clock = std::chrono::system_clock;
00567     auto start = clock::now();
00568 
00569     for (const auto& batch : command_batches) {
00570         std::vector<std::future<void» futures;
00571         is_err.store(false);
00572 
00573         for (const auto& command : batch) {
00574             futures.emplace_back(std::async(std::launch::async, [&command, &final_result, &total_time,
      &result_mutex, &is_err]() {
00575                 if (is_err.load()) {
00576                     return;
00577                 }
00578 
00579                 auto [result, command_time, err] = run_command_par(command);
00580 
00581                 {
00582                     std::lock_guard<std::mutex> lock(result_mutex);
00583                     final_result += result;
00584                     total_time += command_time;
00585                 }
00586 
00587                 if (err) {
00588                     is_err.store(true);
00589                     return;
00590                 }
00591             }));
00592 
00593             if (futures.size() >= num_threads) {
00594                 for (auto& future : futures) {
00595                     future.get();
00596                 }
00597                 futures.clear();
00598             }
00599         }
00600 
00601         for (auto& future : futures) {
00602             future.get();
00603         }
00604 
00605         if (is_err.load()) {
00606             break;
00607         }
00608     }
00609 
00610     auto end = clock::now();
00611     double elapsed = std::chrono::duration<double>(end - start).count();
00612 
00613     std::cout « (is_err.load() ? "\033[31m" : "\033[34m")
00614             « "Compilation process completed in: " « elapsed « " seconds"
00615             « "\033[0m" « std::endl;
00616 
00617     return create_html(final_result, total_time);
00618 }
```

```
00619
00620 #endif // CODERUNNER_HPP
```

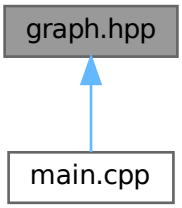
7.5 graph.hpp File Reference

```
#include <iostream>
#include <vector>
#include <unordered_map>
#include <unordered_set>
#include <stack>
#include <queue>
#include <fstream>
#include <cstdlib>
```

Include dependency graph for graph.hpp:



This graph shows which files directly or indirectly include this file:



Classes

- class Graph< T >

  A generic directed graph implementation with utility functions.

7.6 graph.hpp

Go to the documentation of this file.

```
00001 #ifndef GRAPH_HPP
00002 #define GRAPH_HPP
00003
00004 #include <iostream>
00005 #include <vector>
00006 #include <unordered_map>
00007 #include <unordered_set>
00008 #include <stack>
00009 #include <queue>
00010 #include <fstream>
00011 #include <cstdlib>
00012
00022 template <typename T>
```

```
00023 class Graph {
00024 public:
00029     void addNode(const T& node);
00030
00035     void removeNode(const T& node);
00036
00042     void addEdge(const T& from, const T& to);
00043
00049     void removeEdge(const T& from, const T& to);
00050
00055     std::optional<std::vector<T> hasCycle();
00056
00062     std::vector<T> topologicalSort();
00063
00069     int inDegree(const T& node);
00070
00076     int outDegree(const T& node);
00077
00084     void visualize(const std::string& filename, const std::string& imgfilename);
00085
00086 private:
00087
00092     std::unordered_map<T, std::unordered_set<T> adjList;
00093
00103     bool dfsCycleDetection(const T& node, std::unordered_set<T>& visited, std::unordered_set<T>&
      recStack, std::vector<T>& path, std::vector<T>& cycle);
00104
00111     void dfsTopologicalSort(const T& node, std::unordered_set<T>& visited, std::stack<T>& Stack);
00112
00117     void generateDotFile(const std::string& filename);
00118 };
00119
00120 template <typename T>
00121 void Graph<T>::addNode(const T& node) {
00122     adjList[node];
00123 }
00124
00125 template <typename T>
00126 void Graph<T>::removeNode(const T& node) {
00127     adjList.erase(node);
00128     for (auto& [key, neighbors] : adjList) {
00129         neighbors.erase(node);
00130     }
00131 }
00132
00133 template <typename T>
00134 void Graph<T>::addEdge(const T& from, const T& to) {
00135     adjList[from].insert(to);
00136 }
00137
00138 template <typename T>
00139 void Graph<T>::removeEdge(const T& from, const T& to) {
00140     adjList[from].erase(to);
00141 }
00142
00143 template <typename T>
00144 std::optional<std::vector<T> Graph<T>::hasCycle() {
00145     std::unordered_set<T> visited;
00146     std::unordered_set<T> recStack;
00147     std::vector<T> path;
00148     std::vector<T> cycle;
00149     for (const auto& [node, _] : adjList) {
00150         if (dfsCycleDetection(node, visited, recStack, path, cycle)) {
00151             return cycle;
00152         }
00153     }
00154     return std::nullopt;
00155 }
00156
00157 template <typename T>
00158 bool Graph<T>::dfsCycleDetection(const T& node, std::unordered_set<T>& visited, std::unordered_set<T>&
      recStack, std::vector<T>& path, std::vector<T>& cycle) {
00159     if (recStack.find(node) != recStack.end()) {
00160         auto it = std::find(path.begin(), path.end(), node);
00161         cycle.assign(it, path.end());
00162         return true;
00163     }
00164     if (visited.find(node) != visited.end()) {
00165         return false;
00166     }
00167     visited.insert(node);
00168     recStack.insert(node);
00169     path.push_back(node);
00170     for (const auto& neighbor : adjList[node]) {
00171         if (dfsCycleDetection(neighbor, visited, recStack, path, cycle)) {
00172             return true;
00173         }
```

```
00174     }
00175     recStack.erase(node);
00176     path.pop_back();
00177     return false;
00178 }
00179
00180 template <typename T>
00181 std::vector<T> Graph<T>::topologicalSort() {
00182     std::stack<T> Stack;
00183     std::unordered_set<T> visited;
00184     for (const auto& [node, _] : adjList) {
00185         if (visited.find(node) == visited.end()) {
00186             dfsTopologicalSort(node, visited, Stack);
00187         }
00188     }
00189     std::vector<T> result;
00190     while (!Stack.empty()) {
00191         result.push_back(Stack.top());
00192         Stack.pop();
00193     }
00194     return result;
00195 }
00196
00197 template <typename T>
00198 void Graph<T>::dfsTopologicalSort(const T& node, std::unordered_set<T>& visited, std::stack<T>& Stack)
    {
00199     visited.insert(node);
00200     for (const auto& neighbor : adjList[node]) {
00201         if (visited.find(neighbor) == visited.end()) {
00202             dfsTopologicalSort(neighbor, visited, Stack);
00203         }
00204     }
00205     Stack.push(node);
00206 }
00207
00208 template <typename T>
00209 int Graph<T>::inDegree(const T& node) {
00210     int degree = 0;
00211     for (const auto& [key, neighbors] : adjList) {
00212         if (neighbors.find(node) != neighbors.end()) {
00213             degree++;
00214         }
00215     }
00216     return degree;
00217 }
00218
00219 template <typename T>
00220 int Graph<T>::outDegree(const T& node) {
00221     return adjList[node].size();
00222 }
00223
00224 template <typename T>
00225 void Graph<T>::visualize(const std::string& filename, const std::string& imgfilename) {
00226     generateDotFile(filename);
00227     std::string command = "dot -Tpng " + filename + " -o " + imgfilename + ".png";
00228     system(command.c_str());
00229     std::remove(filename.c_str());
00230 }
00231
00232 template <typename T>
00233 void Graph<T>::generateDotFile(const std::string& filename) {
00234     std::ofstream file(filename);
00235     file « "digraph G {\n";
00236     for (const auto& [node, neighbors] : adjList) {
00237         for (const auto& neighbor : neighbors) {
00238             file « "    \"" « node « "\" -> \"" « neighbor « "\";\n";
00239         }
00240     }
00241     file « "}\n";
00242     file.close();
00243 }
00244
00245 #endif // GRAPH_HPP
```

7.7 main.cpp File Reference

```
#include <iostream>
#include <unordered_set>
#include <filesystem>
#include <unordered_map>
#include "parser.hpp"
#include "argparse.hpp"
```

```
#include "graph.hpp"
#include "tabulate.hpp"
#include "coderunner.hpp"
#include "cache.hpp"
```

Include dependency graph for main.cpp:

Classes

- struct Node

  Represents a node in a dependency graph with associated target data.
- struct std::hash< Node >

Namespaces

- namespace std

  Specialization of the `std::hash` template for the `Node` struct.

Typedefs

- using Row_t = tabulate::Table::Row_t

Functions

- void findDependencies (const std::string &target_name, const std::unordered_map< std::string, target > &targ_tab, std::set< std::string > &relevant_targets, const std::string &prefix)

  Recursively finds and collects all dependencies for a given target.
- void parse_and_collect_dependencies (const std::string &filename, const std::string &prefix="")

  Parses a file and collects dependencies, updating global tables and variables.
- void print_banner ()

  Displays a banner with information about the Forge build system.
- int main (int argc, char **argv)

Variables

- std::unordered_map< std::string, std::string > master_var_tab

  table containing variable declarations across all forgefiles
- std::unordered_map< std::string, target > master_targ_tab

  All targets.
- std::unordered_set< std::string > processed_files

  list of files already processed
- std::unordered_set< std::string > concerned_targets

  targets which are concerned based on the target asked to be built
- std::unordered_set< std::string > all_import_vars

  list of all import variables
- std::unordered_map< std::string, std::string > all_import_prefixes

  list of all import variables with their prefixes
- std::unordered_map< std::string, std::string > act_targs

  actual names of targets
- std::set< std::string > all_targs

  list of all targets

7.7.1 Typedef Documentation

7.7.1.1 Row_t

```
using Row_t = tabulate::Table::Row_t
```

Definition at line 12 of file main.cpp.

7.7.2 Function Documentation

7.7.2.1 findDependencies()

```
void findDependencies (
            const std::string & target_name,
            const std::unordered_map< std::string, target > & targ_tab,
            std::set< std::string > & relevant_targets,
            const std::string & prefix)
```

Recursively finds and collects all dependencies for a given target.

This function identifies all the dependencies of a specified target and adds them to the set of relevant targets. It ensures that each target is processed only once by checking if it is already in the set of relevant targets.

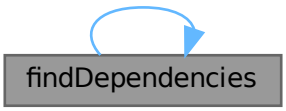
Parameters

target_name	The name of the target whose dependencies are to be found.
targ_tab	A map containing all targets and their associated data, including dependencies.
relevant_targets	A set to store all relevant targets (including dependencies) for the given target.
prefix	A string prefix to prepend to dependency names (if needed).

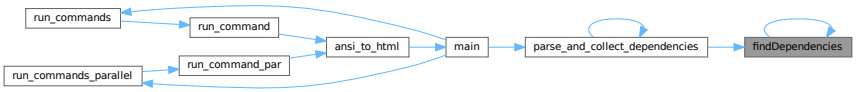
Definition at line 36 of file main.cpp.

```
00036
    {
00037     if (relevant_targets.count(target_name)) {
00038         return;
00039     }
00040
00041     relevant_targets.insert(target_name);
00042     const auto& target = targ_tab.at(target_name);
00043     for (const auto& dep : target.dependencies) {
00044         std::string prefixed_dep = prefix + dep;
00045         if (targ_tab.find(dep) != targ_tab.end()) {
00046             findDependencies(dep, targ_tab, relevant_targets, prefix);
00047         }
00048     }
00049 }
```

Here is the call graph for this function:



Here is the caller graph for this function:



7.7.2.2 main()

```
int main (
            int argc,
            char ** argv)
```

< argument parser object
< argument for loglevel
< file to be considered
< target to be built
< about forge
< number of jobs (parallel threads)
< DAG of the forgefile based on the target and dependencies
< building the target graph
< return if the graph has cycle by printing a cycle as well
< visualises the graph as a png file
< toposort of the graph
< all nodes which are to be compiled
< a cache to store the files which are already compiled
Determines which nodes in the dependency graph need to be compiled.
This loop iterates over the topologically sorted nodes (`topoSort`) and evaluates whether each node requires compilation based on its dependencies and cache status.

- For each node:
  - Retrieves its dependencies.
  - Initializes the `to_compile` status for the node as `false`.
  - Iterates over the dependencies:
    - Checks if the dependency is a target (`act_targs` or `all_targs`).
      - If the dependency is a target and marked for compilation, marks the current node for compilation and adds it to `compilable_nodes`.
    - If the dependency is not a target:
      - Constructs the full file path and checks the cache.
      - If the file is not in the cache, adds it to the cache, marks the current node for compilation, and adds it to `compilable_nodes`.
  - If the node is still not marked for compilation:
    - Constructs the full target path and checks if it exists in the filesystem.
    - If the target does not exist, marks the node for compilation and adds it to `compilable_nodes`.

Note

This logic ensures that only nodes with unmet dependencies or missing targets are marked for compilation, optimizing the build process.

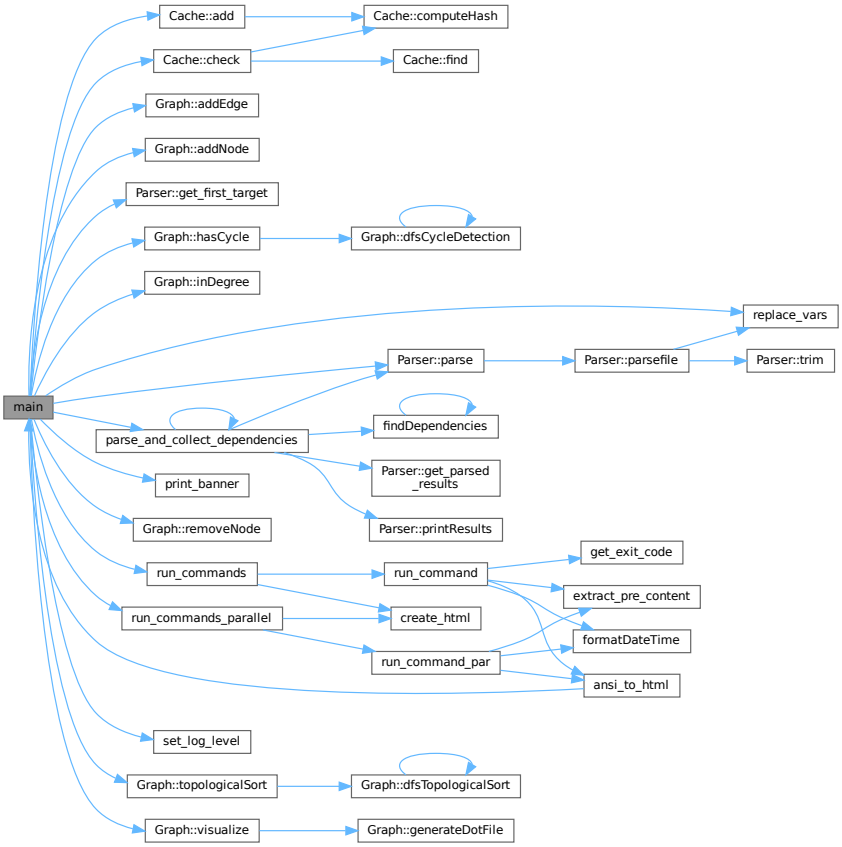
< configure the targets that are to be compiled in parallel
< running jobs in parallel
< running jobs in serial
Definition at line 261 of file main.cpp.

```
00261                                {
00262     argparse::ArgumentParser program("forge");
00263 
00264     program.add_argument("--log-level")
00265       .default_value(std::string{"DEFAULT"})
00266       .help("Specify the Log-Level for Logging. Available Options: DEFAULT, INFO, DEBUG and ERROR");
00267 
00268     program.add_argument("file", "-f", "--file")
00269       .default_value(std::string{"forgefile"})
00270       .help("Specify the file to be considered");
00271 
00272     program.add_argument("target")
00273       .default_value(std::string{""})
00274       .help("Specify the target. If not given, first target will be considered");
00275 
00276     program.add_argument("-a", "--about")
00277       .action([](const auto&){ print_banner(); exit(0);})
00278       .help("Show about information")
00279       .default_value(false)
00280       .implicit_value(true);
00281 
00282     program.add_argument("-j", "--jobs")
00283         .default_value(1)
00284         .help("Specify the number of jobs to run in parallel. Default is 1")
00285         .scan<'i', int>();
00286 
00287     try {
00288       program.parse_args(argc, argv);
00289     } catch (const std::exception &err) {
00290       std::cerr << err.what() << std::endl;
00291       std::cerr << program;
00292       return 1;
00293     }
00294 
00295     std::string loglevel = program.get<std::string>("--log-level");
00296     std::string filename = program.get<std::string>("file");
00297     std::string target = program.get<std::string>("target");
00298 
00299     int njobs = program.get<int>("--jobs");
00300     bool is_par = njobs > 1;
00301 
00302     if(target.length() == 0){
00303       Parser parser(filename);
00304       parser.parse();
00305       target = parser.get_first_target();
00306     }
00307 
00308     concerned_targets.insert(target);
00309 
00310     set_log_level(loglevel);
00311 
00312     LOG(INFO, "Parsing ...\n");
00313 
00314     parse_and_collect_dependencies(filename);
00315 
00316     for (auto& [target, data] : master_targ_tab) {
00317         for (auto& cmd : data.commands) {
00318             replace_vars(cmd, master_var_tab);
00319         }
00320     }
00321 
00322     Graph<Node> targ_graph;
00323 
00324     for (const auto& [target, data] : master_targ_tab) {
00325         Node node{target, data};
00326         targ_graph.addNode(node);
00327     }
00328 
00329     for (const auto& [target, data] : master_targ_tab) {
00330         for (const auto& dep : data.dependencies) {
00331             Node node1{target, data};
00332             if ((master_targ_tab.find(all_import_prefixes[target] + dep) == master_targ_tab.end()) &&
      (master_targ_tab.find(dep) == master_targ_tab.end())) continue;
00333 
00334             std::string prefix;
00335             if (master_targ_tab.find(all_import_prefixes[target] + dep) == master_targ_tab.end()) {
00336                 prefix = "";
00337             } else {
00338                 prefix = all_import_prefixes[target];
00339             }
00340             Node node2{prefix + dep, master_targ_tab[prefix + dep]};
00341             targ_graph.addEdge(node2, node1);
00342         }
00343     }
00344 
00345     auto is_cycle = targ_graph.hasCycle();
00346     if (is_cycle.has_value()) {
```

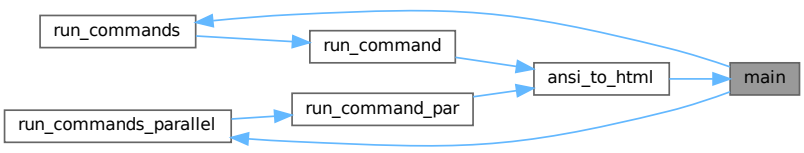
```
00347         std::string cycle_message = "Cycle detected: ";
00348         for (const auto& node : is_cycle.value()) {
00349             cycle_message += node.name + " -> ";
00350         }
00351         cycle_message += is_cycle.value().front().name;
00352         LOG(ERROR, cycle_message);
00353         std::cout << "\n";
00354         return 1;
00355     }
00356 
00357     targ_graph.visualize("graph.dot", "graph");
00358 
00359     auto topoSort = targ_graph.topologicalSort();
00360 
00361     std::vector<Node> compilable_nodes;
00362     Cache cache;
00363     std::unordered_map<std::string, bool> to_compile;
00364 
00365 
00392     for (const auto& node: topoSort) {
00393         auto deps = node.targ_data.dependencies;
00394         to_compile[node.name] = false;
00395 
00396         for (auto& dep: deps) {
00397             bool is_targ = false;
00398             if ((act_targs.find(dep) != act_targs.end()) || (all_targs.find(dep) != all_targs.end()))
      {
00399                 is_targ = true;
00400             }
00401 
00402             if(is_targ) {
00403                 std::string targ_name = dep;
00404                 if (to_compile[targ_name] == true) {
00405                     to_compile[act_targs[node.name]] = true;
00406                     compilable_nodes.push_back(node);
00407                     break;
00408                 }
00409             } else {
00410                 std::string full_file_path = std::filesystem::current_path().string() + "/" + dep;
00411                 if (!cache.check(full_file_path)) {
00412                     cache.add(full_file_path);
00413                     to_compile[act_targs[node.name]] = true;
00414                     compilable_nodes.push_back(node);
00415                     break;
00416                 }
00417             }
00418 
00419         }
00420         if (to_compile[act_targs[node.name]] == false) {
00421             std::string full_target_path = std::filesystem::current_path().string() + "/" +
      act_targs[node.name];
00422             if (!std::filesystem::exists(full_target_path)) {
00423                 to_compile[act_targs[node.name]] = true;
00424                 compilable_nodes.push_back(node);
00425             }
00426         }
00427     }
00428 
00429     if (is_par) {
00430         std::vector<std::vector<std::string» parallelizable_labels;
00431         while (!topoSort.empty()) {
00432             std::vector<std::string> labels;
00433             std::vector<Node> nodesToRemove;
00434             for (const auto& node : topoSort) {
00435                 if (targ_graph.inDegree(node) == 0) {
00436                     if (to_compile[act_targs[node.name]] == true) {
00437                         labels.push_back(node.name);
00438                     }
00439                     nodesToRemove.push_back(node);
00440                 }
00441             }
00442             if(!labels.empty()) {
00443                 parallelizable_labels.push_back(labels);
00444             }
00445             for (const auto& node : nodesToRemove) {
00446                 targ_graph.removeNode(node);
00447             }
00448             topoSort = targ_graph.topologicalSort();
00449         }
00450 
00451         std::vector<std::vector<std::vector<std::string»> parallelizable_commands;
00452         for (const auto& labels : parallelizable_labels) {
00453             std::vector<std::vector<std::string» commands_for_labels;
00454             for (const auto& label : labels) {
00455                 commands_for_labels.push_back(master_targ_tab[label].commands);
00456             }
00457             parallelizable_commands.push_back(commands_for_labels);
```

```
00458         }
00459
00460         std::string output = run_commands_parallel(parallelizable_commands, njobs);
00461         std::ofstream out("forge_output.html");
00462         out « output;
00463         out.close();
00464     } else {
00465         std::vector<std::string> commands;
00466         for (const auto& node : compilable_nodes) {
00467             for (const auto& cmd : node.targ_data.commands) {
00468                 commands.push_back(cmd);
00469             }
00470         }
00471         std::string output = run_commands(commands);
00472         std::ofstream out("forge_output.html");
00473         out « output;
00474         out.close();
00475     }
00476
00477     return 0;
00478 }
```

Here is the call graph for this function:



Here is the caller graph for this function:



7.7.2.3 parse_and_collect_dependencies()

```
void parse_and_collect_dependencies (
            const std::string & filename,
            const std::string & prefix = "")
```

Parses a file and collects dependencies, updating global tables and variables.
This function processes a given file, parses its contents, and collects all relevant dependencies, variables, and targets. It ensures that files are processed only once and handles variable redefinitions across files. The function also recursively processes imported files and their dependencies.

Parameters

filename	The name of the file to parse and process.
prefix	A string prefix to prepend to variable and target names (default is an empty string).

- Updates the `master_var_tab` with variables from the parsed file.
- Updates the `master_targ_tab` with targets from the parsed file.
- Handles recursive imports and ensures no variable redefinitions occur across files.
- Processes dependencies for concerned targets and recursively parses imported files.

Note

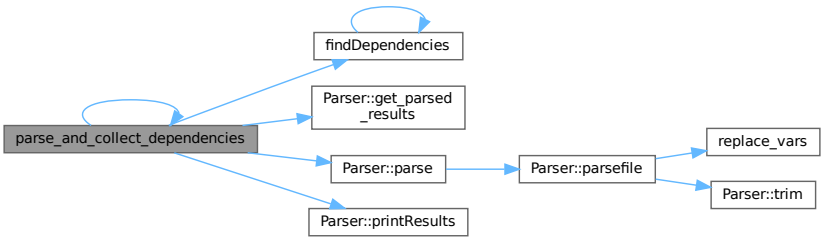
If a variable is redefined across files, the function logs an error and exits.

Definition at line 70 of file main.cpp.

```
00070                                                                                                                {
00071     if (processed_files.find(filename) != processed_files.end()) {
00072         return;
00073     }
00074     processed_files.insert(filename);
00075 
00076     Parser parser(filename);
00077     parser.parse();
00078 
00079     parser.printResults();
00080 
00081     auto [var_tab, targ_tab, import_tab] = parser.get_parsed_results();
00082 
00083     std::set<std::string> relevant_targets;
00084 
00085     for (const auto& target_name : concerned_targets) {
00086         std::string original_target_name = target_name.substr(prefix.length());
00087         if (targ_tab.find(original_target_name) != targ_tab.end()) {
00088             findDependencies(original_target_name, targ_tab, relevant_targets, prefix);
00089         }
00090     }
00091 
00092     for (const auto& [var_name, value] : var_tab) {
00093         master_var_tab[prefix + var_name] = value;
00094     }
00095 
00096     for (const auto& target_name : relevant_targets) {
00097         std::string new_target_name = prefix + target_name;
00098         all_import_prefixes[new_target_name] = prefix;
00099         master_targ_tab[new_target_name] = targ_tab[target_name];
00100         act_targs[new_target_name] = target_name;
00101         all_targs.insert(target_name);
00102     }
00103 
00104     for (auto& [import_var, _] : import_tab) {
00105       if (all_import_vars.count(import_var))
00106       {
00107         LOG(ERROR, " variable redefination error. variable " « import_var « " redefined. Please don't
      reuse variable names across files.")
00108         exit(1);
00109       }
00110       all_import_vars.insert(import_var);
00111     }
00112 
00113     if (relevant_targets.empty()) {
00114         return;
```

```
00115     }
00116
00117     concerned_targets.clear();
00118
00119     for (const auto& target_name : relevant_targets) {
00120         const auto& target = targ_tab[target_name];
00121         for (const auto& dep : target.dependencies) {
00122             for (const auto& [import_var, import_file] : import_tab) {
00123                 if (dep.find(import_var) == 0) {
00124                     concerned_targets.insert(dep);
00125                 }
00126             }
00127         }
00128         for (const auto& dep : target.dependencies) {
00129             for (const auto& [import_var, import_file] : import_tab) {
00130                 if (dep.find(import_var) == 0) {
00131                     parse_and_collect_dependencies(import_file, import_var + ".");
00132                 }
00133             }
00134         }
00135     }
00136 }
```

Here is the call graph for this function:



Here is the caller graph for this function:

7.7.2.4 print_banner()

`void print_banner ()`

Displays a banner with information about the Forge build system.

This function uses the `tabulate` library to create a styled table that displays a banner with details about Forge, including its purpose, license, and usage instructions. The banner is formatted with various font styles, colors, and alignments to enhance readability.

- The banner includes:
  - A title with the name of the build system.
  - A link to the Forge GitHub repository.
  - A brief description of Forge.
  - Highlights such as the required C++ version and license.
  - A quick-start instruction.
- The table is styled using the `tabulate` library with custom colors, font styles, and alignments.

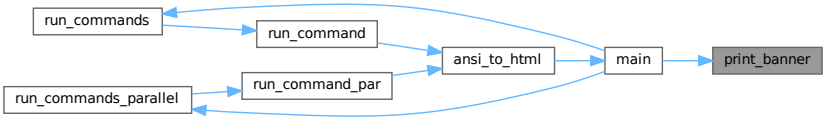
Note

This function outputs the banner directly to the standard output.

Definition at line 228 of file main.cpp.

```
00228                     {
00229   tabulate::Table readme;
00230   readme.format().border_color(tabulate::Color::yellow);
00231
00232   readme.add_row(Row_t{"Forge - an easy and user-friendly build-system"});
00233   readme[0].format().font_style({tabulate::FontStyle::underline, tabulate::FontStyle::bold,
      tabulate::FontStyle::italic}).font_align(tabulate::FontAlign::center).font_color(tabulate::Color::yellow);
00234
00235   readme.add_row(Row_t{"https://github.com/Kronos-192081/Forge"});
00236   readme[1]
00237       .format()
00238       .font_align(tabulate::FontAlign::center)
00239       .font_style({tabulate::FontStyle::underline, tabulate::FontStyle::italic})
00240       .font_color(tabulate::Color::white)
00241       .hide_border_top();
00242
00243   readme.add_row(Row_t{"Forge is a build system built in Modern C++, inspired by Make and designed to
      be more intuitive and user-friendly"});
00244   readme[2].format().font_style({tabulate::FontStyle::italic}).font_color(tabulate::Color::magenta);
00245
00246   tabulate::Table highlights;
00247   highlights.add_row(Row_t{"Requires C++20", "MIT License"});
00248   readme.add_row(Row_t{highlights});
00249   readme[3].format().font_align(tabulate::FontAlign::center).hide_border_top();
00250
00251     readme.add_row(Row_t{"To begin with, run forge -h"});
00252     readme[4]
00253         .format()
00254         .font_align(tabulate::FontAlign::center)
00255         .font_color(tabulate::Color::white)
00256         .hide_border_top();
00257
00258   std::cout << readme << "\n\n";
00259 }
```

Here is the caller graph for this function:



7.7.3 Variable Documentation

7.7.3.1 act_targs

```
std::unordered_map<std::string, std::string> act_targs
```

actual names of targets

Definition at line 20 of file main.cpp.

7.7.3.2 all_import_prefixes

```
std::unordered_map<std::string, std::string> all_import_prefixes
```

list of all import variables with their prefixes

Definition at line 19 of file main.cpp.

7.7.3.3 all_import_vars

```
std::unordered_set<std::string> all_import_vars
```

list of all import variables

Definition at line 18 of file main.cpp.

7.7.3.4 all_targs

`std::set<std::string> all_targs`
list of all targets
Definition at line 21 of file main.cpp.

7.7.3.5 concerned_targets

`std::unordered_set<std::string> concerned_targets`
targets which are concerned based on the target asked to be built
Definition at line 17 of file main.cpp.

7.7.3.6 master_targ_tab

`std::unordered_map<std::string, target> master_targ_tab`
All targets.
Definition at line 15 of file main.cpp.

7.7.3.7 master_var_tab

`std::unordered_map<std::string, std::string> master_var_tab`
table containing variable declarations across all forgefiles
Definition at line 14 of file main.cpp.

7.7.3.8 processed_files

`std::unordered_set<std::string> processed_files`
list of files already processed
Definition at line 16 of file main.cpp.

7.8 main.cpp

Go to the documentation of this file.

```
00001 #include <iostream>
00002 #include <unordered_set>
00003 #include <filesystem>
00004 #include <unordered_map>
00005 #include "parser.hpp"
00006 #include "argparse.hpp"
00007 #include "graph.hpp"
00008 #include "tabulate.hpp"
00009 #include "coderunner.hpp"
00010 #include "cache.hpp"
00011
00012 using Row_t = tabulate::Table::Row_t;
00013
00014 std::unordered_map<std::string, std::string> master_var_tab;
00015 std::unordered_map<std::string, target> master_targ_tab;
00016 std::unordered_set<std::string> processed_files;
00017 std::unordered_set<std::string> concerned_targets;
00018 std::unordered_set<std::string> all_import_vars;
00019 std::unordered_map<std::string, std::string> all_import_prefixes;
00020 std::unordered_map<std::string, std::string> act_targs;
00021 std::set<std::string> all_targs;
00022
00023
00036 void findDependencies(const std::string& target_name, const std::unordered_map<std::string, target>&
      targ_tab, std::set<std::string>& relevant_targets, const std::string& prefix) {
00037     if (relevant_targets.count(target_name)) {
00038         return;
00039     }
00040
00041     relevant_targets.insert(target_name);
00042     const auto& target = targ_tab.at(target_name);
00043     for (const auto& dep : target.dependencies) {
00044         std::string prefixed_dep = prefix + dep;
00045         if (targ_tab.find(dep) != targ_tab.end()) {
00046             findDependencies(dep, targ_tab, relevant_targets, prefix);
00047         }
00048     }
00049 }
```

```
00050
00070 void parse_and_collect_dependencies(const std::string& filename, const std::string& prefix = "") {
00071     if (processed_files.find(filename) != processed_files.end()) {
00072         return;
00073     }
00074     processed_files.insert(filename);
00075
00076     Parser parser(filename);
00077     parser.parse();
00078
00079     parser.printResults();
00080
00081     auto [var_tab, targ_tab, import_tab] = parser.get_parsed_results();
00082
00083     std::set<std::string> relevant_targets;
00084
00085     for (const auto& target_name : concerned_targets) {
00086         std::string original_target_name = target_name.substr(prefix.length());
00087         if (targ_tab.find(original_target_name) != targ_tab.end()) {
00088             findDependencies(original_target_name, targ_tab, relevant_targets, prefix);
00089         }
00090     }
00091
00092     for (const auto& [var_name, value] : var_tab) {
00093         master_var_tab[prefix + var_name] = value;
00094     }
00095
00096     for (const auto& target_name : relevant_targets) {
00097         std::string new_target_name = prefix + target_name;
00098         all_import_prefixes[new_target_name] = prefix;
00099         master_targ_tab[new_target_name] = targ_tab[target_name];
00100         act_targs[new_target_name] = target_name;
00101         all_targs.insert(target_name);
00102     }
00103
00104     for (auto& [import_var, _] : import_tab) {
00105       if (all_import_vars.count(import_var))
00106       {
00107         LOG(ERROR, " variable redefination error. variable " « import_var « " redefined. Please don't
      reuse variable names across files.")
00108         exit(1);
00109       }
00110       all_import_vars.insert(import_var);
00111     }
00112
00113     if (relevant_targets.empty()) {
00114         return;
00115     }
00116
00117     concerned_targets.clear();
00118
00119     for (const auto& target_name : relevant_targets) {
00120         const auto& target = targ_tab[target_name];
00121         for (const auto& dep : target.dependencies) {
00122             for (const auto& [import_var, import_file] : import_tab) {
00123                 if (dep.find(import_var) == 0) {
00124                     concerned_targets.insert(dep);
00125                 }
00126             }
00127         }
00128         for (const auto& dep : target.dependencies) {
00129             for (const auto& [import_var, import_file] : import_tab) {
00130                 if (dep.find(import_var) == 0) {
00131                     parse_and_collect_dependencies(import_file, import_var + ".");
00132                 }
00133             }
00134         }
00135     }
00136 }
00137
00145 struct Node {
00146     std::string name;
00147     target targ_data;
00148
00154     bool operator==(const Node& other) const {
00155         return name == other.name;
00156     }
00157
00163     bool operator!=(const Node& other) const {
00164         return !(*this == other);
00165     }
00166
00172     bool operator<(const Node& other) const {
00173         return name < other.name;
00174     }
00175
00182     friend std::ostream& operator«(std::ostream& os, const Node& node) {
```

```
00183         os « node.name;
00184         return os;
00185     }
00186 };
00187
00195 namespace std {
00196     template <>
00197     struct hash<Node> {
00204         std::size_t operator()(const Node& node) const {
00205             return std::hash<std::string>()(node.name);
00206         }
00207     };
00208 }
00209
00228 void print_banner() {
00229   tabulate::Table readme;
00230   readme.format().border_color(tabulate::Color::yellow);
00231
00232   readme.add_row(Row_t{"Forge - an easy and user-friendly build-system"});
00233   readme[0].format().font_style({tabulate::FontStyle::underline, tabulate::FontStyle::bold,
      tabulate::FontStyle::italic}).font_align(tabulate::FontAlign::center).font_color(tabulate::Color::yellow);
00234
00235   readme.add_row(Row_t{"https://github.com/Kronos-192081/Forge"});
00236   readme[1]
00237       .format()
00238       .font_align(tabulate::FontAlign::center)
00239       .font_style({tabulate::FontStyle::underline, tabulate::FontStyle::italic})
00240       .font_color(tabulate::Color::white)
00241       .hide_border_top();
00242
00243   readme.add_row(Row_t{"Forge is a build system built in Modern C++, inspired by Make and designed to
      be more intuitive and user-friendly"});
00244   readme[2].format().font_style({tabulate::FontStyle::italic}).font_color(tabulate::Color::magenta);
00245
00246   tabulate::Table highlights;
00247   highlights.add_row(Row_t{"Requires C++20", "MIT License"});
00248   readme.add_row(Row_t{highlights});
00249   readme[3].format().font_align(tabulate::FontAlign::center).hide_border_top();
00250
00251     readme.add_row(Row_t{"To begin with, run forge -h"});
00252     readme[4]
00253         .format()
00254         .font_align(tabulate::FontAlign::center)
00255         .font_color(tabulate::Color::white)
00256         .hide_border_top();
00257
00258   std::cout « readme « "\n\n";
00259 }
00260
00261 int main(int argc, char ** argv) {
00262     argparse::ArgumentParser program("forge");
00263
00264     program.add_argument("--log-level")
00265       .default_value(std::string{"DEFAULT"})
00266       .help("Specify the Log-Level for Logging. Available Options: DEFAULT, INFO, DEBUG and ERROR");
00267
00268     program.add_argument("file", "-f", "--file")
00269       .default_value(std::string{"forgefile"})
00270       .help("Specify the file to be considered");
00271
00272     program.add_argument("target")
00273       .default_value(std::string{""})
00274       .help("Specify the target. If not given, first target will be considered");
00275
00276     program.add_argument("-a", "--about")
00277       .action([](const auto&){ print_banner(); exit(0);})
00278       .help("Show about information")
00279       .default_value(false)
00280       .implicit_value(true);
00281
00282     program.add_argument("-j", "--jobs")
00283         .default_value(1)
00284         .help("Specify the number of jobs to run in parallel. Default is 1")
00285         .scan<'i', int>();
00286
00287     try {
00288       program.parse_args(argc, argv);
00289     } catch (const std::exception &err) {
00290       std::cerr « err.what() « std::endl;
00291       std::cerr « program;
00292       return 1;
00293    }
00294
00295     std::string loglevel = program.get<std::string>("--log-level");
00296     std::string filename = program.get<std::string>("file");
00297     std::string target = program.get<std::string>("target");
00298
```

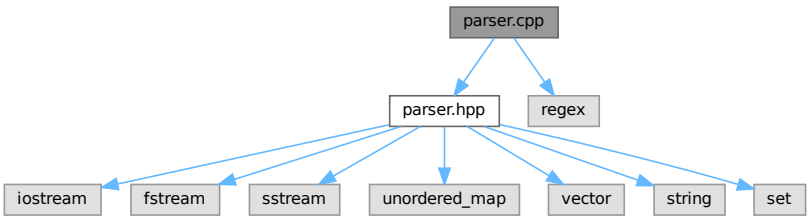
```
00299     int njobs = program.get<int>("--jobs");
00300     bool is_par = njobs > 1;
00301 
00302     if(target.length() == 0){
00303       Parser parser(filename);
00304       parser.parse();
00305       target = parser.get_first_target();
00306     }
00307 
00308     concerned_targets.insert(target);
00309 
00310     set_log_level(loglevel);
00311 
00312     LOG(INFO, "Parsing ...\n");
00313 
00314     parse_and_collect_dependencies(filename);
00315 
00316     for (auto& [target, data] : master_targ_tab) {
00317         for (auto& cmd : data.commands) {
00318             replace_vars(cmd, master_var_tab);
00319         }
00320     }
00321 
00322     Graph<Node> targ_graph;
00323 
00324     for (const auto& [target, data] : master_targ_tab) {
00325         Node node{target, data};
00326         targ_graph.addNode(node);
00327     }
00328 
00329     for (const auto& [target, data] : master_targ_tab) {
00330         for (const auto& dep : data.dependencies) {
00331             Node node1{target, data};
00332             if ((master_targ_tab.find(all_import_prefixes[target] + dep) == master_targ_tab.end()) &&
    (master_targ_tab.find(dep) == master_targ_tab.end())) continue;
00333 
00334             std::string prefix;
00335             if (master_targ_tab.find(all_import_prefixes[target] + dep) == master_targ_tab.end()) {
00336                 prefix = "";
00337             } else {
00338                 prefix = all_import_prefixes[target];
00339             }
00340             Node node2{prefix + dep, master_targ_tab[prefix + dep]};
00341             targ_graph.addEdge(node2, node1);
00342         }
00343     }
00344 
00345     auto is_cycle = targ_graph.hasCycle();
00346     if (is_cycle.has_value()) {
00347         std::string cycle_message = "Cycle detected: ";
00348         for (const auto& node : is_cycle.value()) {
00349             cycle_message += node.name + " -> ";
00350         }
00351         cycle_message += is_cycle.value().front().name;
00352         LOG(ERROR, cycle_message);
00353         std::cout << "\n";
00354         return 1;
00355     }
00356 
00357     targ_graph.visualize("graph.dot", "graph");
00358 
00359     auto topoSort = targ_graph.topologicalSort();
00360 
00361     std::vector<Node> compilable_nodes;
00362     Cache cache;
00363     std::unordered_map<std::string, bool> to_compile;
00364 
00365 
00392     for (const auto& node: topoSort) {
00393         auto deps = node.targ_data.dependencies;
00394         to_compile[node.name] = false;
00395 
00396         for (auto& dep: deps) {
00397             bool is_targ = false;
00398             if ((act_targs.find(dep) != act_targs.end()) || (all_targs.find(dep) != all_targs.end()))
    {
00399                 is_targ = true;
00400             }
00401 
00402             if(is_targ) {
00403                 std::string targ_name = dep;
00404                 if (to_compile[targ_name] == true) {
00405                     to_compile[act_targs[node.name]] = true;
00406                     compilable_nodes.push_back(node);
00407                     break;
00408                 }
00409             } else {
```

```
00410                 std::string full_file_path = std::filesystem::current_path().string() + "/" + dep;
00411                 if (!cache.check(full_file_path)) {
00412                     cache.add(full_file_path);
00413                     to_compile[act_targs[node.name]] = true;
00414                     compilable_nodes.push_back(node);
00415                     break;
00416                 }
00417             }
00418 
00419         }
00420         if (to_compile[act_targs[node.name]] == false) {
00421             std::string full_target_path = std::filesystem::current_path().string() + "/" +
      act_targs[node.name];
00422             if (!std::filesystem::exists(full_target_path)) {
00423                 to_compile[act_targs[node.name]] = true;
00424                 compilable_nodes.push_back(node);
00425             }
00426         }
00427     }
00428 
00429     if (is_par) {
00430         std::vector<std::vector<std::string» parallelizable_labels;
00431         while (!topoSort.empty()) {
00432             std::vector<std::string> labels;
00433             std::vector<Node> nodesToRemove;
00434             for (const auto& node : topoSort) {
00435                 if (targ_graph.inDegree(node) == 0) {
00436                     if (to_compile[act_targs[node.name]] == true) {
00437                         labels.push_back(node.name);
00438                     }
00439                     nodesToRemove.push_back(node);
00440                 }
00441             }
00442             if(!labels.empty()) {
00443                 parallelizable_labels.push_back(labels);
00444             }
00445             for (const auto& node : nodesToRemove) {
00446                 targ_graph.removeNode(node);
00447             }
00448             topoSort = targ_graph.topologicalSort();
00449         }
00450 
00451         std::vector<std::vector<std::vector<std::string»> parallelizable_commands;
00452         for (const auto& labels : parallelizable_labels) {
00453             std::vector<std::vector<std::string» commands_for_labels;
00454             for (const auto& label : labels) {
00455                 commands_for_labels.push_back(master_targ_tab[label].commands);
00456             }
00457             parallelizable_commands.push_back(commands_for_labels);
00458         }
00459 
00460         std::string output = run_commands_parallel(parallelizable_commands, njobs);
00461         std::ofstream out("forge_output.html");
00462         out « output;
00463         out.close();
00464     } else {
00465         std::vector<std::string> commands;
00466         for (const auto& node : compilable_nodes) {
00467             for (const auto& cmd : node.targ_data.commands) {
00468                 commands.push_back(cmd);
00469             }
00470         }
00471         std::string output = run_commands(commands);
00472         std::ofstream out("forge_output.html");
00473         out « output;
00474         out.close();
00475     }
00476 
00477     return 0;
00478 }
```

7.9 parser.cpp File Reference

```
#include "parser.hpp"
#include <regex>
```

Include dependency graph for parser.cpp:



Functions

- void set_log_level (std::string loglevel)

  Sets the log level for filtering log messages.
- void replace_vars (std::string &line, const std::unordered_map< std::string, std::string > &variables)

  Replaces variables in a string with their corresponding values.

Variables

- LogLevel FILTER_LEVEL = DEFAULT

  Global variable to filter log messages based on their level.

7.9.1 Function Documentation

7.9.1.1 replace_vars()

```
void replace_vars (
            std::string & line,
            const std::unordered_map< std::string, std::string > & variables)
```

Replaces variables in a string with their corresponding values.

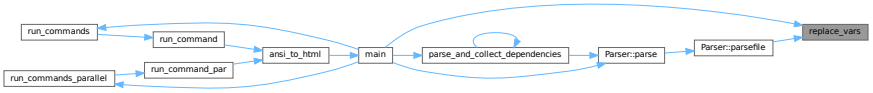
Parameters

line	The string in which variables will be replaced.
variables	A map of variable names to their values.

Definition at line 20 of file parser.cpp.

```
00020                                                                                                                {
00021     std::regex var_pattern(R"(\$\(([\w\.]+)\))");
00022     std::smatch match;
00023     std::string::const_iterator search_start(line.cbegin());
00024     while (std::regex_search(search_start, line.cend(), match, var_pattern)) {
00025         std::string var_name = match[1].str();
00026         if (variables.find(var_name) != variables.end()) {
00027             line.replace(match.position(0), match.length(0), variables.at(var_name));
00028             search_start = line.cbegin() + match.position(0) + variables.at(var_name).length();
00029         } else {
00030             search_start = match.suffix().first;
00031         }
00032     }
00033 }
```

Here is the caller graph for this function:



7.9.1.2 set_log_level()

```
void set_log_level (
            std::string loglevel)
```

Sets the log level for filtering log messages.

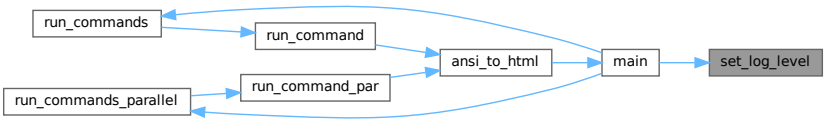
Parameters

loglevel	The log level as a string (e.g., "INFO", "DEBUG").

Definition at line 6 of file parser.cpp.

```
00006                                       {
00007     if (loglevel == "INFO") FILTER_LEVEL = INFO;
00008     else if (loglevel == "DEBUG") FILTER_LEVEL = DEBUG;
00009     else if (loglevel == "ERROR") FILTER_LEVEL= ERROR;
00010     else FILTER_LEVEL = DEFAULT;
00011 }
```

Here is the caller graph for this function:



7.9.2 Variable Documentation

7.9.2.1 FILTER_LEVEL

```
LogLevel FILTER_LEVEL = DEFAULT
```

Global variable to filter log messages based on their level.
Definition at line 4 of file parser.cpp.

7.10 parser.cpp

Go to the documentation of this file.

```
00001 #include "parser.hpp"
00002 #include <regex>
00003
00004 LogLevel FILTER_LEVEL = DEFAULT;
00005
00006 void set_log_level(std::string loglevel) {
00007     if (loglevel == "INFO") FILTER_LEVEL = INFO;
00008     else if (loglevel == "DEBUG") FILTER_LEVEL = DEBUG;
00009     else if (loglevel == "ERROR") FILTER_LEVEL= ERROR;
00010     else FILTER_LEVEL = DEFAULT;
00011 }
00012
00013 void Parser::trim(std::string &s) {
00014         if(s[0] == '\t') return;
00015         size_t start = s.find_first_not_of(" \t");
00016         size_t end = s.find_last_not_of(" \t");
00017         s = (start == std::string::npos) ? "" : s.substr(start, end - start + 1);
00018 }
00019
00020 void replace_vars(std::string &line, const std::unordered_map<std::string, std::string> &variables) {
00021     std::regex var_pattern(R"(\$\(([\w\.]+)\))");
00022     std::smatch match;
00023     std::string::const_iterator search_start(line.cbegin());
00024     while (std::regex_search(search_start, line.cend(), match, var_pattern)) {
00025         std::string var_name = match[1].str();
00026         if (variables.find(var_name) != variables.end()) {
00027             line.replace(match.position(0), match.length(0), variables.at(var_name));
00028             search_start = line.cbegin() + match.position(0) + variables.at(var_name).length();
00029         } else {
00030             search_start = match.suffix().first;
00031         }
```

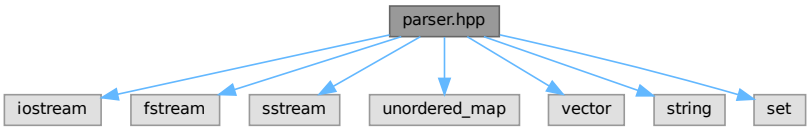
```
00032     }
00033 }
00034
00035 void Parser::parsefile(std::ifstream &file) {
00036     bool is_first_target = true;
00037     std::string line;
00038     bool line_cont = false;
00039     while (std::getline(file, line)) {
00040         line_no++;
00041         trim(line);
00042
00043         // print the exact characters of the line in ascii
00044         if (line.empty() || line[0] == '#') continue;
00045         replace_vars(line, variables);
00046
00047         if (line[0] == '\t') {
00048             if (line.length() == 1) continue;
00049             if (!currentTarget.empty()) {
00050                 if (!line.empty() && line.back() == '\\') {
00051                     line.pop_back();
00052                     std::string last_cmd = "";
00053                     if (line_cont) {
00054                         last_cmd = targets[currentTarget].commands.back();
00055                         targets[currentTarget].commands.pop_back();
00056                     }
00057                     last_cmd += line.substr(1);
00058                     targets[currentTarget].commands.push_back(last_cmd);
00059                     line_cont = true;
00060                     continue;
00061                 }
00062
00063                 if (line_cont){
00064                     std::string last_cmd = targets[currentTarget].commands.back();
00065                     targets[currentTarget].commands.pop_back();
00066                     last_cmd += line.substr(1);
00067                     targets[currentTarget].commands.push_back(last_cmd);
00068                     line_cont = false;
00069                 } else {
00070                     targets[currentTarget].commands.push_back(line.substr(1));
00071                 }
00072             }
00073         } else if (line.find(":") != std::string::npos &&
00074                     (line.find("=") == std::string::npos || line.find(":") < line.find("="))) {
00075
00076             size_t colonPos = line.find(":");
00077             currentTarget = line.substr(0, colonPos);
00078             trim(currentTarget);
00079             if(is_first_target) {
00080                 first_target = currentTarget;
00081                 is_first_target = false;
00082             }
00083             std::istringstream depStream(line.substr(colonPos + 1));
00084             std::string dep;
00085             while (depStream » dep) {
00086                 trim(dep);
00087                 targets[currentTarget].dependencies.insert(dep);
00088             }
00089         } else if (line.find("=") != std::string::npos) {
00090             size_t eqPos = line.find("=");
00091             std::string var = line.substr(0, eqPos);
00092             std::string value = line.substr(eqPos + 1);
00093             trim(var);
00094             trim(value);
00095             std::string after_assign = line.substr(eqPos + 1, 7);
00096             trim(after_assign);
00097             if (after_assign == "import") {
00098                 std::string import_str = line.substr(eqPos + 8);
00099                 trim(import_str);
00100                 import_str = import_str.substr(1, import_str.size()- 2);
00101                 imports[var] = import_str;
00102             } else {
00103                 variables[var] = value;
00104             }
00105         } else {
00106             LOG(ERROR, " Error parsing forgefile in line " « line_no « ": " « line);
00107             exit(0);
00108         }
00109     }
00110 }
00111
00112 void Parser::printResults() const {
00113     LOG(INFO, "Variables:\n");
00114     for (const auto &var : variables) {
00115         LOG(INFO, var.first « " = " « var.second « "\n");
00116     }
00117
00118     LOG(INFO, "\nTargets:\n");
```

```
00119     for (const auto &t : targets) {
00120         LOG(INFO, t.first « ": ");
00121         for (const auto &dep : t.second.dependencies) {
00122             LOG(INFO, dep « " ");
00123         }
00124         LOG(INFO, "\nCommands:\n");
00125         for (const auto &cmd : t.second.commands) {
00126             LOG(INFO, cmd « "\n");
00127         }
00128 
00129         LOG(INFO, std::endl);
00130     }
00131 
00132     LOG(INFO, "\nImports:\n");
00133     for (const auto &imp : imports) {
00134         LOG(INFO, imp.first « " = import " « imp.second « "\n");
00135     }
00136 }
```

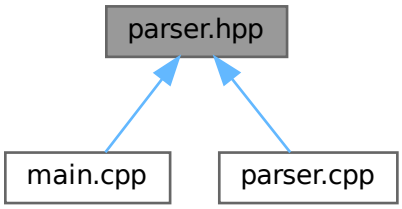
7.11 parser.hpp File Reference

```
#include <iostream>
#include <fstream>
#include <sstream>
#include <unordered_map>
#include <vector>
#include <string>
#include <set>
```

Include dependency graph for parser.hpp:



This graph shows which files directly or indirectly include this file:



Classes

- struct target

  Struct representing a build target with its dependencies and commands.
- class Parser

  Class for parsing a build configuration file.

Macros

- #define COLOR_RESET "\033[0m"
- #define COLOR_RED "\033[31m"
- #define COLOR_YELLOW "\033[33m"
- #define COLOR_BLUE "\033[34m"
- #define LOG_COLOR(LOG_LEVEL)

  Macro to determine the color for log messages based on the log level.
- #define LOG(LOG_LEVEL, LOG_STRING)

  Macro to log messages to the console with color and log level filtering.

Typedefs

- using var_table = std::unordered_map<std::string, std::string>

  Alias for a variable table.
- using targ_table = std::unordered_map<std::string, target>

  Alias for a target table.
- using import_table = std::unordered_map<std::string, std::string>

  Alias for an import table.

Enumerations

- enum LogLevel { INFO , DEBUG , DEFAULT , ERROR }

  Enum representing different log levels.

Functions

- void set_log_level (std::string loglevel)

  Sets the log level for filtering log messages.
- void replace_vars (std::string &line, const std::unordered_map< std::string, std::string > &variables)

  Replaces variables in a string with their corresponding values.

Variables

- LogLevel FILTER_LEVEL

  Global variable to filter log messages based on their level.

7.11.1 Macro Definition Documentation

7.11.1.1 COLOR_BLUE

```
#define COLOR_BLUE "\033[34m"
```

Definition at line 15 of file parser.hpp.

7.11.1.2 COLOR_RED

```
#define COLOR_RED "\033[31m"
```

Definition at line 13 of file parser.hpp.

7.11.1.3 COLOR_RESET

```
#define COLOR_RESET "\033[0m"
```

Definition at line 12 of file parser.hpp.

7.11.1.4 COLOR_YELLOW

```
#define COLOR_YELLOW "\033[33m"
```

Definition at line 14 of file parser.hpp.

7.11.1.5 LOG

```
#define LOG(
                LOG_LEVEL,
                LOG_STRING)
```

Value:

```
    if (LOG_LEVEL >= FILTER_LEVEL) {                                          \
        std::cout « LOG_COLOR(LOG_LEVEL) « "[" « #LOG_LEVEL « "] "       \
                  « LOG_STRING « COLOR_RESET « std::endl;                  \
    }
```

Macro to log messages to the console with color and log level filtering.
This macro checks if the provided log level is greater than or equal to the global `FILTER_LEVEL`. If so, it prints the log message to the console with the appropriate color and log level tag.

Parameters

LOG_LEVEL	The log level of the message (e.g., INFO, DEBUG, ERROR).
LOG_STRING	The log message to be printed.

- The log message is prefixed with the log level (e.g., `[INFO]`, `[DEBUG]`).
- The message is displayed in a color corresponding to the log level.
- The color is reset after the message is printed.

Definition at line 75 of file parser.hpp.

```
00075 #define LOG(LOG_LEVEL, LOG_STRING)                                          \
00076     if (LOG_LEVEL >= FILTER_LEVEL) {                                       \
00077         std::cout « LOG_COLOR(LOG_LEVEL) « "[" « #LOG_LEVEL « "] "       \
00078                   « LOG_STRING « COLOR_RESET « std::endl;                  \
00079     }
```

7.11.1.6 LOG_COLOR

```
#define LOG_COLOR(
                LOG_LEVEL)
```

Value:

```
(LOG_LEVEL == INFO ? COLOR_BLUE : (LOG_LEVEL == DEBUG ? COLOR_YELLOW : COLOR_RED))
```

Macro to determine the color for log messages based on the log level.
This macro evaluates the log level and returns the corresponding color code:

- `INFO` logs are displayed in blue.
- `DEBUG` logs are displayed in yellow.
- Other log levels (e.g., `ERROR`) are displayed in red.

Parameters

LOG_LEVEL	The log level (e.g., INFO, DEBUG, ERROR).

Returns

The color code as a string.

Definition at line 58 of file parser.hpp.

7.11.2 Typedef Documentation

7.11.2.1 import_table

```
using import_table = std::unordered_map<std::string, std::string>
```

Alias for an import table.
Definition at line 118 of file parser.hpp.

7.11.2.2 targ_table

```
using targ_table = std::unordered_map<std::string, target>
```
Alias for a target table.
Definition at line 117 of file parser.hpp.

7.11.2.3 var_table

```
using var_table = std::unordered_map<std::string, std::string>
```
Alias for a variable table.
Definition at line 116 of file parser.hpp.

7.11.3 Enumeration Type Documentation

7.11.3.1 LogLevel

```
enum LogLevel
```
Enum representing different log levels.

Enumerator

INFO	Informational messages.
DEBUG	Debugging messages.
DEFAULT	Default log level.
ERROR	Error messages.

Definition at line 20 of file parser.hpp.

```
00020             {
00021     INFO,
00022     DEBUG,
00023     DEFAULT,
00024     ERROR
00025 };
```

7.11.4 Function Documentation

7.11.4.1 replace_vars()

```
void replace_vars (
            std::string & line,
            const std::unordered_map< std::string, std::string > & variables)
```
Replaces variables in a string with their corresponding values.

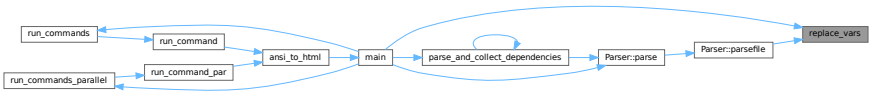
Parameters

line	The string in which variables will be replaced.
variables	A map of variable names to their values.

Definition at line 20 of file parser.cpp.

```
00020                                                                                                   {
00021     std::regex var_pattern(R"(\$\(([\w\.]+)\))");
00022     std::smatch match;
00023     std::string::const_iterator search_start(line.cbegin());
00024     while (std::regex_search(search_start, line.cend(), match, var_pattern)) {
00025         std::string var_name = match[1].str();
00026         if (variables.find(var_name) != variables.end()) {
00027             line.replace(match.position(0), match.length(0), variables.at(var_name));
00028             search_start = line.cbegin() + match.position(0) + variables.at(var_name).length();
00029         } else {
00030             search_start = match.suffix().first;
00031         }
00032     }
00033 }
```

Here is the caller graph for this function:



7.11.4.2 set_log_level()

```
void set_log_level (
            std::string loglevel)
```

Sets the log level for filtering log messages.

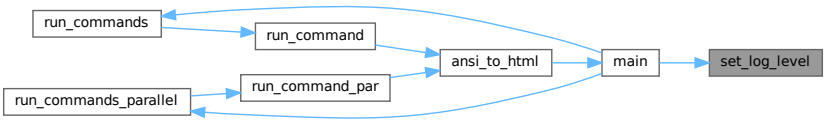
Parameters

loglevel	The log level as a string (e.g., "INFO", "DEBUG").

Definition at line 6 of file parser.cpp.

```
00006                                          {
00007     if (loglevel == "INFO") FILTER_LEVEL = INFO;
00008     else if (loglevel == "DEBUG") FILTER_LEVEL = DEBUG;
00009     else if (loglevel == "ERROR") FILTER_LEVEL= ERROR;
00010     else FILTER_LEVEL = DEFAULT;
00011 }
```

Here is the caller graph for this function:



7.11.5 Variable Documentation

7.11.5.1 FILTER_LEVEL

```
LogLevel FILTER_LEVEL  [extern]
```

Global variable to filter log messages based on their level.
Definition at line 4 of file parser.cpp.

7.12 parser.hpp

Go to the documentation of this file.

```
00001 #ifndef PARSER_H
00002 #define PARSER_H
00003
00004 #include <iostream>
00005 #include <fstream>
00006 #include <sstream>
00007 #include <unordered_map>
00008 #include <vector>
00009 #include <string>
00010 #include <set>
00011
00012 #define COLOR_RESET   "\033[0m"
00013 #define COLOR_RED     "\033[31m"
00014 #define COLOR_YELLOW  "\033[33m"
00015 #define COLOR_BLUE    "\033[34m"
00016
```

```
00020 enum LogLevel{
00021     INFO,
00022     DEBUG,
00023     DEFAULT,
00024     ERROR
00025 };
00026
00030 extern LogLevel FILTER_LEVEL;
00031
00037 void set_log_level(std::string loglevel);
00038
00045 void replace_vars(std::string &line, const std::unordered_map<std::string, std::string> &variables);
00046
00058 #define LOG_COLOR(LOG_LEVEL) (LOG_LEVEL == INFO ? COLOR_BLUE : (LOG_LEVEL == DEBUG ? COLOR_YELLOW :
      COLOR_RED))
00059
00075 #define LOG(LOG_LEVEL, LOG_STRING)                                                    \
00076     if (LOG_LEVEL >= FILTER_LEVEL) {                                                  \
00077         std::cout « LOG_COLOR(LOG_LEVEL) « "[" « #LOG_LEVEL « "] "       \
00078                   « LOG_STRING « COLOR_RESET « std::endl;                  \
00079     }
00080
00084 struct target {
00085     std::set<std::string> dependencies;
00086     std::vector<std::string> commands;
00087
00095     friend std::ostream& operator«(std::ostream& os, const target& targ) {
00096         os « "Dependencies[";
00097         for (auto dep : targ.dependencies) {
00098             os « dep;
00099             if (dep != *targ.dependencies.rbegin()) {
00100                 os « ", ";
00101             }
00102         }
00103         os « "] Commands[";
00104         for (size_t i = 0; i < targ.commands.size(); ++i) {
00105             os « targ.commands[i];
00106             if (i != targ.commands.size() - 1) {
00107                 os « ", ";
00108             }
00109         }
00110         os « "]";
00111
00112         return os;
00113     }
00114 };
00115
00116 using var_table = std::unordered_map<std::string, std::string>;
00117 using targ_table = std::unordered_map<std::string, target>;
00118 using import_table = std::unordered_map<std::string, std::string>;
00119
00123 class Parser {
00124     public:
00125
00131         Parser(const std::string& file): filename(file) {}
00132
00136         void printResults() const;
00137
00143         std::tuple<var_table, targ_table, import_table> get_parsed_results() {
00144             return std::make_tuple(std::move(variables), std::move(targets), std::move(imports));
00145         }
00146
00152         void parse() {
00153             std::ifstream file(filename);
00154             if (!file) {
00155                 std::cerr « "Error opening forgefile. No such file found." « std::endl;
00156                 exit(1);
00157             }
00158             parsefile(file);
00159         }
00160
00166         void set_file_name(std::string& name) {
00167             filename = name;
00168         }
00169
00175         std::string get_first_target() { return first_target; }
00176
00177     private:
00178         std::string filename;
00179         var_table variables;
00180         targ_table targets;
00181         import_table imports;
00182
00183         std::string first_target;
00184         std::string currentTarget;
00185         int line_no = 0;
00186
```

```
00192         void trim(std::string &s);
00193
00199         void parsefile(std::ifstream &file);
00200 };
00201
00202 #endif
```

7.13 README.md File Reference