Middle East Technical University Northern Cyprus Campus
Computer Engineering Program

CNG491 Computer Engineering Design I

Markopy Documentation

Ata Hakçıl - 2243467
Osman Ömer Yıldıztugay - 1921956
Celal Sahir Çetiner - 1755420
Yunus Emre Yılmaz - 2243723

Supervised by
Assoc. Prof. Dr. Okan Topçu

0.4.1 Documentation

Chapter 1

Markov Passwords

Table of Contents

Markov Passwords

Generate wordlists with markov models.
Wiki · Complete documentation · Report Bug · Add a Bug

<details open="open">

</details>

1.1 About The Project

This project aims to generate wordlists using markov models.

1.1.1 Built With

- CPP, with dependecies: boost, python3-dev, QT-5.

1.2 Getting Started

If you'd just like to use the project without contributing, check out the releases page. If you want to build, check out wiki for building the project.

1.2.1 Prerequisites

1.2.1.0.1 MarkovModel

- Make for linux, Visual Studio/MSBuild for Windows.

1.2.1.0.2 MarkovPasswords

- Boost.ProgramOptions (tested on 1.76.0)

1.2.1.0.3 Markopy

- Boost.Python (tested on 1.76.0)
- Python development package (tested on python 3.8)

1.2.1.0.4 MarkovPasswordsGUI

- QT development environment.

1.2.2 Installing Dependencies

1.2.2.0.1 Windows

- QT: Install `QT For Windows`
- Boost:
  - Download Boost from `its website`
  - Unzip the contents.
  - Launch "Visual Studio Developer Command Prompt"
  - Move to the boost installation directory. Run `bootstrap.bat`
  - Run `b2`.
- Python: You can use the windows app store to download python runtime and libraries.

1.2.2.0.2 Linux

- QT: Follow `this guide` to install QT on Linux.
- Boost: run `sudo apt-get install libboost-all-dev`
- Python: run `sudo apt-get install python3`

1.2.3 Installation

See the Wiki Page

1.2.4 Building

Building process can be fairly complicated depending on the environment.

1.3 Linux

If you've set up the dependencies, you can just build the project with make. List of directives is below.

```
.PHONY: all
all: model mp
model: $(INCLUDE)/$(MM_LIB)
mp: $(BIN)/$(MP_EXEC)
markopy: $(BIN)/$(MPY_SO)
.PHONY: clean
clean:
    $(RM) -r $(BIN)/*
```

1.4 Windows

Set up correct environment variables for BOOST_ROOT% (folder containing boost, libs, stage, tools) and PYTH↩ ON_PATH% (folder containing include, lib, libs, Tools, python.exe/python3.exe).
If you've set up the dependencies and environment variables correctly, you can open the solution with Visual Studio and build with that.

1.5 Known Common issues

1.5.1 Linux

1.5.1.1 Markopy - Python.h - Not found

Make sure you have the development version of python package, which includes the required header files. Check if header files exist: `/usr/include/python*`
If it doesn't, run `sudo apt-get install python3-dev`

1.5.1.2 Markopy/MarkovPasswords - *.so not found, or other library related issues when building

Run `ls /usr/lib/x86_64-linux-gnu/ | grep boost` and check the shared object filenames. A common issue is that lboost is required but filenames are formatted as llibboost, or vice versa.
Do the same for python related library issues, run: `ls /usr/lib/x86_64-linux-gnu/ | grep python` to verify filename format is as required.
If not, you can modify the makefile, or create symlinks such as: `ln -s /usr/lib/x86_64-linux-gnu/libboost↩ _python38.so /usr/lib/x86_64-linux-gnu/boost_python38.so`

1.5.2 Windows

1.5.2.1 Boost - Bootstrap.bat "ctype.h" not found

- Make sure you are working in the "Visual Studio Developer Command Prompt" terminal.
- Make sure you have Windows 10 SDK installed.
- From VS developer terminal, run echo INCLUDE%. If result does not have the windows sdk folders, run the following before running bootstrap (change your sdk version instead of 10.0.19041.0):

```
set INCLUDE=%INCLUDE%;C:\Program Files (x86)\Windows Kits\NETFXSDK\4.8\include\um;C:\Program Files
 (x86)\Windows Kits\10\include\10.0.19041.0\ucrt;C:\Program Files (x86)\Windows
 Kits\10\include\10.0.19041.0\shared;C:\Program Files (x86)\Windows
 Kits\10\include\10.0.19041.0\um;C:\Program Files (x86)\Windows
 Kits\10\include\10.0.19041.0\winrt;C:\Program Files (x86)\Windows
 Kits\10\include\10.0.19041.0\cppwinrt
set LIB=%LIB%;C:\Program Files (x86)\Windows Kits\10\lib\10.0.19041.0\ucrt\x64;C:\Program Files
 (x86)\Windows Kits\10\lib\10.0.19041.0\um\x64
```

1.5.2.2 Cannot open file "*.lib"

Make sure you have set the BOOST_ROOT environment variable correctly. Make sure you ran `b2` to build library files from boost sources.

1.5.2.3 Python.h not found

Make sure you have python installed, and make sure you set PYTHON_PATH environment variable.

1.5.2.4 Simplified Theory

What is a markov model Below, is the example Markov Model which can generate strings with the alphabet "a,b,c"

Iteration 1 Below is a demonstration of how training will be done. For this example, we are going to adjust the model with string "ab", and our occurrence will be "3" From MarkovPasswords, inside the train function, Model↩ ::adjust is called with "ab" and "3" parameters.

Now, Model::adjust will iteratively adjust the edge weights accordingly. It starts by adjusting weight between start and "a" node. This is done by calling Edge::adjust of the edge between the nodes.
After adjustment, ajust function iterates to the next character, "b", and does the same thing.
As this string is finished, it will adjust the final weight, b->"end"

Iteration 2 This time, same procedure will be applied for "bacb" string, with occurrence value of 12.

Iteration 38271 As the model is trained, hidden linguistical patterns start to appear, and our model looks like this With our dataset, without doing any kind of linugistic analysis ourselves, our Markov Model has highlighted that strings are more likely to start with a, b tends to follow a, and a is likely to be repeated in the string.

1.5.3 Contributing

Feel free to contribute.

1.5.4 Contact

Twitter - `@ahakcil`

Chapter 2

Deprecated List

Member Markov::API::MarkovPasswords::Generate (unsigned long int n, const char *wordlistFileName, int minLen=6, int maxLen=12, int threads=20)

See Markov::API::MatrixModel::FastRandomWalk for more information.

Chapter 3

Namespace Index

3.1 Namespace List

Here is a list of all namespaces with brief descriptions:

Chapter 4

Hierarchical Index

4.1 Class Hierarchy

This inheritance list is sorted roughly, but not completely, alphabetically:

Chapter 5

Class Index

5.1 Class List

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

Chapter 6

File Index

6.1 File List

Here is a list of all files with brief descriptions:

Chapter 7

Namespace Documentation

7.1 markopy_cli Namespace Reference

Functions

- def cli_init (input_model)
- def cli_train (model, dataset, seperator, output, output_forced=False, bulk=False)
- def cli_generate (model, wordlist, bulk=False)

Variables

- parser
- help
- default
- action
- args = parser.parse_args()
- corpus_list = os.listdir(args.dataset)
- def model = cli_init(args.input)
- output_file_name = corpus
- string model_extension = ""
- output_forced
- True
- bulk
- model_list = os.listdir(args.input)
- model_base = input
- output

7.1.1 Detailed Description

```
@namespace Markov::Markopy::Python
```

7.1.2 Function Documentation

7.1.2.1 cli_generate()

```
def markopy_cli.cli_generate (
              model,
              wordlist,
              bulk = False )
```

Definition at line 114 of file markopy_cli.py.

```
00114 def cli_generate(model, wordlist, bulk=False):
00115     if not (wordlist or args.count):
00116         logging.pprint("Generation mode requires -w/--wordlist and -n/--count parameters. Exiting.")
00117         exit(2)
00118
00119     if (bulk and os.path.isfile(wordlist)):
00120         logging.pprint(f"{wordlist} exists and will be overwritten.", 1)
00121     model.Generate(int(args.count), wordlist, int(args.min), int(args.max), int(args.threads))
00122
00123
```

7.1.2.2 cli_init()

```
def markopy_cli.cli_init (
              input_model )
```

Definition at line 61 of file markopy_cli.py.

```
00061 def cli_init(input_model):
00062     logging.VERBOSITY = 0
00063     if args.verbosity:
00064         logging.VERBOSITY = args.verbosity
00065         logging.pprint(f"Verbosity set to {args.verbosity}.", 2)
00066
00067     logging.pprint("Initializing model.", 1)
00068     model = markopy.MarkovPasswords()
00069     logging.pprint("Model initialized.", 2)
00070
00071     logging.pprint("Importing model file.", 1)
00072
00073     if (not os.path.isfile(input_model)):
00074         logging.pprint(f"Model file at {input_model} not found. Check the file path, or working
      directory")
00075         exit(1)
00076
00077     model.Import(input_model)
00078     logging.pprint("Model imported successfully.", 2)
00079     return model
00080
00081
```

7.1.2.3 cli_train()

```
def markopy_cli.cli_train (
              model,
              dataset,
              seperator,
              output,
              output_forced = False,
              bulk = False )
```

Definition at line 82 of file markopy_cli.py.

```
00082 def cli_train(model, dataset, seperator, output, output_forced=False, bulk=False):
00083     if not (dataset and seperator and (output or not output_forced)):
00084         logging.pprint(
00085             f"Training mode requires -d/--dataset{', -o/--output' if output_forced else ''} and
      -s/--seperator parameters. Exiting.")
00086         exit(2)
00087
00088     if (not bulk and not os.path.isfile(dataset)):
00089         logging.pprint(f"{dataset} doesn't exists. Check the file path, or working directory")
00090         exit(3)
00091
00092     if (output and os.path.isfile(output)):
00093         logging.pprint(f"{output} exists and will be overwritten.", 1)
00094
00095     if (seperator == '\\t'):
00096         logging.pprint("Escaping seperator.", 3)
00097         seperator = '\t'
00098
00099     if (len(seperator) != 1):
00100         logging.pprint(f'Delimiter must be a single character, and "{seperator}" is not accepted.')
00101         exit(4)
00102
00103     logging.pprint(f'Starting training.', 3)
00104     model.Train(dataset, seperator, int(args.threads))
00105     logging.pprint(f'Training completed.', 2)
00106
```

```
00107     if (output):
00108         logging.pprint(f'Exporting model to {output}', 2)
00109         model.Export(output)
00110     else:
00111         logging.pprint(f'Model will not be exported.', 1)
00112
00113
```

7.1.3 Variable Documentation

7.1.3.1 action

`markopy_cli.action`
Definition at line 49 of file markopy_cli.py.

7.1.3.2 args

`markopy_cli.args = parser.parse_args()`
Definition at line 58 of file markopy_cli.py.

7.1.3.3 bulk

`markopy_cli.bulk`
Definition at line 139 of file markopy_cli.py.

7.1.3.4 corpus_list

`markopy_cli.corpus_list = os.listdir(args.dataset)`
Definition at line 130 of file markopy_cli.py.

7.1.3.5 default

`markopy_cli.default`
Definition at line 41 of file markopy_cli.py.

7.1.3.6 help

`markopy_cli.help`
Definition at line 27 of file markopy_cli.py.

7.1.3.7 model

`def markopy_cli.model = cli_init(args.input)`
Definition at line 132 of file markopy_cli.py.

7.1.3.8 model_base

`markopy_cli.model_base = input`
Definition at line 153 of file markopy_cli.py.

7.1.3.9 model_extension

```
markopy_cli.model_extension = ""
```
Definition at line 135 of file markopy_cli.py.

7.1.3.10 model_list

```
markopy_cli.model_list = os.listdir(args.input)
```
Definition at line 147 of file markopy_cli.py.

7.1.3.11 output

```
markopy_cli.output
```
Definition at line 167 of file markopy_cli.py.

7.1.3.12 output_file_name

```
markopy_cli.output_file_name = corpus
```
Definition at line 134 of file markopy_cli.py.

7.1.3.13 output_forced

```
markopy_cli.output_forced
```
Definition at line 139 of file markopy_cli.py.

7.1.3.14 parser

```
markopy_cli.parser
```
Initial value:
```
00001 =  argparse.ArgumentParser(description="Python wrapper for MarkovPasswords.",
00002                            epilog=f, formatter_class=argparse.RawTextHelpFormatter)
```
Definition at line 12 of file markopy_cli.py.

7.1.3.15 True

```
markopy_cli.True
```
Definition at line 139 of file markopy_cli.py.

7.2 Markov Namespace Reference

Namespace for the markov-model related classes. Contains Model, Node and Edge classes.

Namespaces

- API

  Namespace for the MarkovPasswords API.
- GUI

  namespace for MarkovPasswords API GUI wrapper
- Markopy
- Random

  Objects related to RNG.

Classes

- class Edge

  Edge class used to link nodes in the model together.
- class Model

  class for the final Markov Model, constructed from nodes and edges.
- class Node

  A node class that for the vertices of model. Connected with eachother using Edge.

7.2.1 Detailed Description

Namespace for the markov-model related classes. Contains Model, Node and Edge classes.

7.3 Markov::API Namespace Reference

Namespace for the MarkovPasswords API.

Namespaces

- CLI

  Structure to hold parsed cli arguements.
- Concurrency

  Namespace for Concurrency related classes.
- CUDA

  Namespace for objects requiring CUDA libraries.

Classes

- class MarkovPasswords

  Markov::Model with char represented nodes.
- class ModelMatrix

  Class to flatten and reduce Markov::Model to a Matrix.

7.3.1 Detailed Description

Namespace for the MarkovPasswords API.

7.4 Markov::API::CLI Namespace Reference

Structure to hold parsed cli arguements.

Classes

- struct _programOptions

  Structure to hold parsed cli arguements.
- class Argparse

  Parse command line arguements.
- class Terminal

  pretty colors for Terminal. Windows Only.

Typedefs

- typedef struct Markov::API::CLI::_programOptions ProgramOptions

  Structure to hold parsed cli arguements.

Functions

- std::ostream & operator<< (std::ostream &os, const Markov::API::CLI::Terminal::color &c)

7.4.1 Detailed Description

Structure to hold parsed cli arguements.
Namespace for the CLI objects

7.4.2 Typedef Documentation

7.4.2.1 ProgramOptions

```
typedef struct Markov::API::CLI::_programOptions Markov::API::CLI::ProgramOptions
```

Structure to hold parsed cli arguements.

7.4.3 Function Documentation

7.4.3.1 operator<<()

```
std::ostream& Markov::API::CLI::operator<< (
            std::ostream & os,
            const Markov::API::CLI::Terminal::color & c )
```

overload for std::cout.
Definition at line 60 of file term.cpp.

```
00060                                                                {
00061     char buf[6];
00062     sprintf(buf,"%d",Terminal::colormap.find(c)->second);
00063     os « "\e[1;" « buf « "m";
00064     return os;
00065 }
```

References Markov::API::CLI::Terminal::colormap.

7.5 Markov::API::Concurrency Namespace Reference

Namespace for Concurrency related classes.

Classes

- class ThreadSharedListHandler

  Simple class for managing shared access to file.

7.5.1 Detailed Description

Namespace for Concurrency related classes.

7.6 Markov::API::CUDA Namespace Reference

Namespace for objects requiring CUDA libraries.

Namespaces

- Random

  *Namespace for Random engines operable under **device** space.*

Classes

- class CUDADeviceController

  Controller class for CUDA device.
- class CUDAModelMatrix

  Extension of Markov::API::ModelMatrix which is modified to run on GPU devices.

Functions

- __global__ void FastRandomWalkCUDAKernel (unsigned long int n, int minLen, int maxLen, char ∗outputBuffer, char ∗matrixIndex, long int ∗totalEdgeWeights, long int ∗valueMatrix, char ∗edgeMatrix, int matrixSize, int memoryPerKernelGrid, unsigned long ∗seed)

  CUDA kernel for the FastRandomWalk operation.
- __device__ char ∗ strchr (char ∗p, char c, int s_len)

  *srtchr implementation on **device** space*

7.6.1 Detailed Description

Namespace for objects requiring CUDA libraries.

7.6.2 Function Documentation

7.6.2.1 FastRandomWalkCUDAKernel()

```
__global__ void Markov::API::CUDA::FastRandomWalkCUDAKernel (
            unsigned long int n,
            int minLen,
            int maxLen,
            char * outputBuffer,
            char * matrixIndex,
            long int * totalEdgeWeights,
            long int * valueMatrix,
            char * edgeMatrix,
            int matrixSize,
            int memoryPerKernelGrid,
            unsigned long * seed )
```

CUDA kernel for the FastRandomWalk operation.
Will be initiated by CPU and continued by GPU (**global** tag)

Parameters

n	- Number of passwords to generate.
minlen	- minimum string length for a single generation
maxLen	- maximum string length for a single generation
outputBuffer	- VRAM ptr to the output buffer
matrixIndex	- VRAM ptr to the matrix indices
totalEdgeWeights	- VRAM ptr to the totalEdgeWeights array
valueMatrix	- VRAM ptr to the edge weights array
edgeMatrix	- VRAM ptr to the edge representations array
matrixSize	- Size of the matrix dimensions
memoryPerKernelGrid	- Maximum memory usage per kernel grid
seed	- seed chunk to generate the random from (generated & used by Marsaglia)

7.6.2.2 strchr()

```
__device__ char* Markov::API::CUDA::strchr (
            char * p,
            char c,
            int s_len )
```

srtchr implementation on **device** space
Fint the first matching index of a string

Parameters

p	- string to check
c	- character to match
s_len	- maximum string length

Returns

pointer to the match

7.7 Markov::API::CUDA::Random Namespace Reference

Namespace for Random engines operable under **device** space.

Classes

- class Marsaglia

  *Extension of Markov::Random::Marsaglia which is capable o working on **device** space.*

Functions

- __device__ unsigned long devrandom (unsigned long &x, unsigned long &y, unsigned long &z)

  *Marsaglia Random Generation function operable in **device** space.*

7.7.1 Detailed Description

Namespace for Random engines operable under **device** space.

7.7.2 Function Documentation

7.7.2.1 devrandom()

```
__device__ unsigned long Markov::API::CUDA::Random::devrandom (
            unsigned long & x,
            unsigned long & y,
            unsigned long & z )
```

Marsaglia Random Generation function operable in **device** space.

Parameters

x	marsaglia internal x. Not constant, (ref)
y	marsaglia internal y. Not constant, (ref)
z	marsaglia internal z. Not constant, (ref)

Returns

returns z

Definition at line 43 of file cudarandom.h.

```
00043                                                                                {
00044         unsigned long t;
00045         x ^= x << 16;
00046         x ^= x >> 5;
00047         x ^= x << 1;
00048 
00049         t = x;
00050         x = y;
00051         y = z;
00052         z = t ^ x ^ y;
00053 
00054         return z;
00055     }
```

7.8 Markov::GUI Namespace Reference

namespace for MarkovPasswords API GUI wrapper

Classes

- class about

  QT Class for about page.
- class CLI

  QT CLI Class.
- class MarkovPasswordsGUI

  Reporting UI.
- class menu

  QT Menu class.
- class Train

  QT Training page class.

7.8.1 Detailed Description

namespace for MarkovPasswords API GUI wrapper

7.9 Markov::Markopy Namespace Reference

Functions

- BOOST_PYTHON_MODULE (markopy)

7.9.1 Function Documentation

7.9.1.1 BOOST_PYTHON_MODULE()

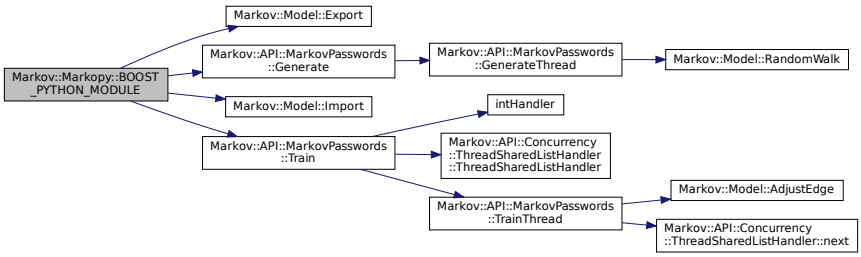
```
Markov::Markopy::BOOST_PYTHON_MODULE (
            markopy  )
```

Definition at line 11 of file markopy.cpp.

```
00012     {
00013         bool (Markov::API::MarkovPasswords::*Import)(const char*) = &Markov::Model<char>::Import;
00014         bool (Markov::API::MarkovPasswords::*Export)(const char*) = &Markov::Model<char>::Export;
00015         class_<Markov::API::MarkovPasswords>("MarkovPasswords", init<>())
00016             .def(init<>())
00017             .def("Train", &Markov::API::MarkovPasswords::Train,
00018             "Train the model\n"
00019             "\n"
00020             ":param datasetFileName: Ifstream* to the dataset. If null, use class member\n"
```

```
00021             ":param delimiter: a character, same as the delimiter in dataset content\n"
00022             ":param threads: number of OS threads to spawn\n")
00023             .def("Generate", &Markov::API::MarkovPasswords::Generate,
00024             "Generate passwords from a trained model.\n"
00025             ":param n: Ifstream* to the dataset. If null, use class member\n"
00026             ":param wordlistFileName: a character, same as the delimiter in dataset content\n"
00027             ":param minLen: number of OS threads to spawn\n"
00028             ":param maxLen: Ifstream* to the dataset. If null, use class member\n"
00029             ":param threads: a character, same as the delimiter in dataset content\n"
00030             ":param threads: number of OS threads to spawn\n")
00031             .def("Import", Import, "Import a model file.")
00032             .def("Export", Export, "Export a model to file.")
00033         ;
00034     };
```

References Markov::Model< NodeStorageType >::Export(), Markov::API::MarkovPasswords::Generate(), Markov::Model< NodeStor
and Markov::API::MarkovPasswords::Train().
Here is the call graph for this function:



7.10 Markov::Random Namespace Reference

Objects related to RNG.

Classes

- class DefaultRandomEngine

  Implementation using Random.h default random engine.
- class Marsaglia

  Implementation of Marsaglia Random Engine.
- class Mersenne

  Implementation of Mersenne Twister Engine.
- class RandomEngine

  An abstract class for Random Engine.

7.10.1 Detailed Description

Objects related to RNG.

7.11 model_2gram Namespace Reference

Variables

- alphabet = string.printable

  password alphabet
- f = open('../../models/2gram.mdl', "wb")

  output file handle

7.11.1 Detailed Description

```
python script for generating a 2gram model
```

7.11.2 Variable Documentation

7.11.2.1 alphabet

```
model_2gram.alphabet = string.printable
```
password alphabet
Definition at line 10 of file model_2gram.py.

7.11.2.2 f

```
model_2gram.f = open('../../models/2gram.mdl', "wb")
```
output file handle
Definition at line 16 of file model_2gram.py.

7.12 random Namespace Reference

7.12.1 Detailed Description

-model

```
python script for generating a 2gram model
```

7.13 random-model Namespace Reference

Variables

- alphabet = string.printable
  password alphabet
- f = open('../../models/random.mdl', "wb")
  output file handle

7.13.1 Variable Documentation

7.13.1.1 alphabet

```
random-model.alphabet = string.printable
```
password alphabet
Definition at line 10 of file random-model.py.

7.13.1.2 f

```
random-model.f = open('../../models/random.mdl', "wb")
```
output file handle
Definition at line 16 of file random-model.py.

7.14 Testing Namespace Reference

Namespace for Microsoft Native Unit Testing Classes.

Namespaces

- MarkovModel

  Testing namespace for MarkovModel.
- MarkovPasswords

  Testing namespace for MarkovPasswords.
- MVP

  Testing Namespace for Minimal Viable Product.

7.14.1 Detailed Description

Namespace for Microsoft Native Unit Testing Classes.

7.15 Testing::MarkovModel Namespace Reference

Testing namespace for MarkovModel.

Functions

- TEST_CLASS (Edge)

  Test class for rest of Edge cases.
- TEST_CLASS (Node)

  Test class for rest of Node cases.
- TEST_CLASS (Model)

  Test class for rest of model cases.

7.15.1 Detailed Description

Testing namespace for MarkovModel.

7.15.2 Function Documentation

7.15.2.1 TEST_CLASS() [1/3]

```
Testing::MarkovModel::TEST_CLASS (
            Edge  )
```

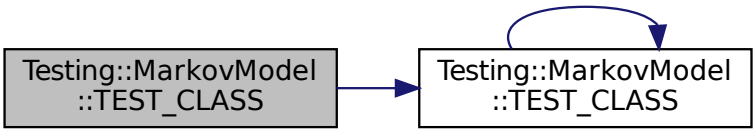
Test class for rest of Edge cases.
send exception on integer underflow
test integer overflows
Definition at line 494 of file UnitTests.cpp.

```
00495         {
00496         public:
00499             TEST_METHOD(except_integer_underflow) {
00500                 auto _underflow_adjust = [] {
00501                     Markov::Node<unsigned char>* LeftNode = new Markov::Node<unsigned char>('l');
00502                     Markov::Node<unsigned char>* RightNode = new Markov::Node<unsigned char>('r');
00503                     Markov::Edge<unsigned char>* e = new Markov::Edge<unsigned char>(LeftNode,
      RightNode);
00504                     e->AdjustEdge(15);
00505                     e->AdjustEdge(-30);
00506                     delete LeftNode;
00507                     delete RightNode;
00508                     delete e;
00509                 };
00510                 Assert::ExpectException<std::underflow_error>(_underflow_adjust);
```

```
00511             }
00512
00515             TEST_METHOD(except_integer_overflow) {
00516                 auto _overflow_adjust = [] {
00517                     Markov::Node<unsigned char>* LeftNode = new Markov::Node<unsigned char>('l');
00518                     Markov::Node<unsigned char>* RightNode = new Markov::Node<unsigned char>('r');
00519                     Markov::Edge<unsigned char>* e = new Markov::Edge<unsigned char>(LeftNode,
      RightNode);
00520                     e->AdjustEdge(~0ull);
00521                     e->AdjustEdge(1);
00522                     delete LeftNode;
00523                     delete RightNode;
00524                     delete e;
00525                 };
00526                 Assert::ExpectException<std::underflow_error>(_overflow_adjust);
00527             }
00528         };
```

References TEST_CLASS().

Here is the call graph for this function:



7.15.2.2 TEST_CLASS() [2/3]

```
Testing::MarkovModel::TEST_CLASS (
            Model  )
```

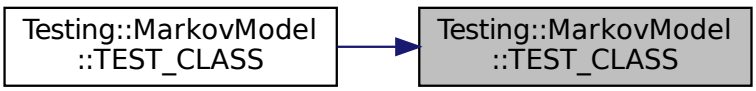
Test class for rest of model cases.

Definition at line 592 of file UnitTests.cpp.

```
00593         {
00594         public:
00595             TEST_METHOD(functional_random_walk) {
00596                 unsigned char* res2 = new unsigned char[12 + 5];
00597                 Markov::Random::Marsaglia MarsagliaRandomEngine;
00598                 Markov::Model<unsigned char> m;
00599                 Markov::Node<unsigned char>* starter = m.StarterNode();
00600                 Markov::Node<unsigned char>* a = new Markov::Node<unsigned char>('a');
00601                 Markov::Node<unsigned char>* b = new Markov::Node<unsigned char>('b');
00602                 Markov::Node<unsigned char>* c = new Markov::Node<unsigned char>('c');
00603                 Markov::Node<unsigned char>* end = new Markov::Node<unsigned char>(0xff);
00604                 starter->Link(a)->AdjustEdge(1);
00605                 a->Link(b)->AdjustEdge(1);
00606                 b->Link(c)->AdjustEdge(1);
00607                 c->Link(end)->AdjustEdge(1);
00608
00609                 char* res = (char*)m.RandomWalk(&MarsagliaRandomEngine,1,12,res2);
00610                 Assert::IsFalse(strcmp(res, "abc"));
00611             }
00612             TEST_METHOD(functionoal_random_walk_without_any) {
00613                 Markov::Model<unsigned char> m;
00614                 Markov::Node<unsigned char>* starter = m.StarterNode();
00615                 Markov::Node<unsigned char>* a = new Markov::Node<unsigned char>('a');
00616                 Markov::Node<unsigned char>* b = new Markov::Node<unsigned char>('b');
00617                 Markov::Node<unsigned char>* c = new Markov::Node<unsigned char>('c');
00618                 Markov::Node<unsigned char>* end = new Markov::Node<unsigned char>(0xff);
00619                 Markov::Edge<unsigned char>* res = NULL;
00620                 starter->Link(a)->AdjustEdge(1);
00621                 a->Link(b)->AdjustEdge(1);
00622                 b->Link(c)->AdjustEdge(1);
00623                 c->Link(end)->AdjustEdge(1);
00624
00625                 res = starter->FindEdge('D');
00626                 Assert::IsNull(res);
00627
00628             }
00629         };
```

References TEST_CLASS().
Referenced by TEST_CLASS().
Here is the call graph for this function:

Here is the caller graph for this function:



7.15.2.3 TEST_CLASS() [3/3]

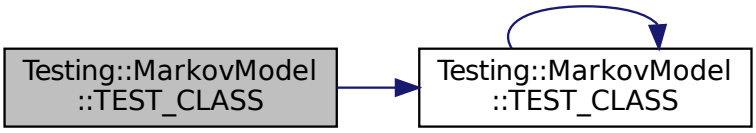
```
Testing::MarkovModel::TEST_CLASS (
            Node  )
```

Test class for rest of Node cases.
test RandomNext with 64 bit high values
test RandomNext with 64 bit high values
randomNext when no edges are present
Definition at line 532 of file UnitTests.cpp.

```
00533         {
00534         public:
00535
00538             TEST_METHOD(rand_next_u64) {
00539                 Markov::Random::Marsaglia MarsagliaRandomEngine;
00540                 Markov::Node<unsigned char>* src = new Markov::Node<unsigned char>('a');
00541                 Markov::Node<unsigned char>* target1 = new Markov::Node<unsigned char>('b');
00542                 Markov::Edge<unsigned char>* e = src->Link(target1);
00543                 e->AdjustEdge((unsigned long)(1ull << 63));
00544                 Markov::Node<unsigned char>* res = src->RandomNext(&MarsagliaRandomEngine);
00545                 Assert::IsTrue(res == target1);
00546                 delete src;
00547                 delete target1;
00548                 delete e;
00549
00550             }
00551
00554             TEST_METHOD(rand_next_u64_max) {
00555                 Markov::Random::Marsaglia MarsagliaRandomEngine;
00556                 Markov::Node<unsigned char>* src = new Markov::Node<unsigned char>('a');
00557                 Markov::Node<unsigned char>* target1 = new Markov::Node<unsigned char>('b');
00558                 Markov::Edge<unsigned char>* e = src->Link(target1);
00559                 e->AdjustEdge((0xffffFFFF));
00560                 Markov::Node<unsigned char>* res = src->RandomNext(&MarsagliaRandomEngine);
00561                 Assert::IsTrue(res == target1);
00562                 delete src;
00563                 delete target1;
```

```
00564                 delete e;
00565
00566             }
00567
00570             TEST_METHOD(uninitialized_rand_next) {
00571
00572                 auto _invalid_next = [] {
00573                     Markov::Random::Marsaglia MarsagliaRandomEngine;
00574                     Markov::Node<unsigned char>* src = new Markov::Node<unsigned char>('a');
00575                     Markov::Node<unsigned char>* target1 = new Markov::Node<unsigned char>('b');
00576                     Markov::Edge<unsigned char>* e = new Markov::Edge<unsigned char>(src, target1);
00577                     Markov::Node<unsigned char>* res = src->RandomNext(&MarsagliaRandomEngine);
00578
00579                     delete src;
00580                     delete target1;
00581                     delete e;
00582                 };
00583
00584                 Assert::ExpectException<std::logic_error>(_invalid_next);
00585             }
00586
00587
00588         };
```

References TEST_CLASS().
Here is the call graph for this function:



7.16 Testing::MarkovPasswords Namespace Reference

Testing namespace for MarkovPasswords.

7.16.1 Detailed Description

Testing namespace for MarkovPasswords.

7.17 Testing::MVP Namespace Reference

Testing Namespace for Minimal Viable Product.

Namespaces

- MarkovModel

  Testing Namespace for MVP MarkovModel.
- MarkovPasswords

  Testing namespace for MVP MarkovPasswords.

7.17.1 Detailed Description

Testing Namespace for Minimal Viable Product.

7.18 Testing::MVP::MarkovModel Namespace Reference

Testing Namespace for MVP MarkovModel.

Functions

- TEST_CLASS (Edge)

  Test class for minimal viable Edge.
- TEST_CLASS (Node)

  Test class for minimal viable Node.
- TEST_CLASS (Model)

  Test class for minimal viable Model.

7.18.1 Detailed Description

Testing Namespace for MVP MarkovModel.

7.18.2 Function Documentation

7.18.2.1 TEST_CLASS() [1/3]

```
Testing::MVP::MarkovModel::TEST_CLASS (
            Edge  )
```

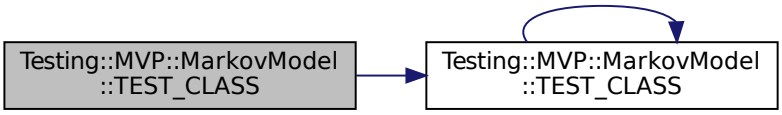
Test class for minimal viable Edge.
test default constructor
test linked constructor with two nodes
test AdjustEdge function
test TraverseNode returning RightNode
test LeftNode/RightNode setter
test negative adjustments
Definition at line 21 of file UnitTests.cpp.

```
00022             {
00023             public:
00024
00027                 TEST_METHOD(default_constructor) {
00028                     Markov::Edge<unsigned char>* e = new Markov::Edge<unsigned char>;
00029                     Assert::IsNull(e->LeftNode());
00030                     Assert::IsNull(e->RightNode());
00031                     delete e;
00032                 }
00033
00036                 TEST_METHOD(linked_constructor) {
00037                     Markov::Node<unsigned char>* LeftNode = new Markov::Node<unsigned char>('l');
00038                     Markov::Node<unsigned char>* RightNode = new Markov::Node<unsigned char>('r');
00039                     Markov::Edge<unsigned char>* e = new Markov::Edge<unsigned char>(LeftNode,
      RightNode);
00040                     Assert::IsTrue(LeftNode == e->LeftNode());
00041                     Assert::IsTrue(RightNode == e->RightNode());
00042                     delete LeftNode;
00043                     delete RightNode;
00044                     delete e;
00045                 }
00046
00049                 TEST_METHOD(AdjustEdge) {
00050                     Markov::Node<unsigned char>* LeftNode = new Markov::Node<unsigned char>('l');
00051                     Markov::Node<unsigned char>* RightNode = new Markov::Node<unsigned char>('r');
00052                     Markov::Edge<unsigned char>* e = new Markov::Edge<unsigned char>(LeftNode,
      RightNode);
00053                     e->AdjustEdge(15);
00054                     Assert::AreEqual(15ull, e->EdgeWeight());
00055                     e->AdjustEdge(15);
00056                     Assert::AreEqual(30ull, e->EdgeWeight());
00057                     delete LeftNode;
00058                     delete RightNode;
00059                     delete e;
00060                 }
00061
00064                 TEST_METHOD(TraverseNode) {
00065                     Markov::Node<unsigned char>* LeftNode = new Markov::Node<unsigned char>('l');
00066                     Markov::Node<unsigned char>* RightNode = new Markov::Node<unsigned char>('r');
00067                     Markov::Edge<unsigned char>* e = new Markov::Edge<unsigned char>(LeftNode,
      RightNode);
00068                     Assert::IsTrue(RightNode == e->TraverseNode());
00069                     delete LeftNode;
00070                     delete RightNode;
```

```
00071                     delete e;
00072                 }
00073
00076                 TEST_METHOD(set_left_and_right) {
00077                     Markov::Node<unsigned char>* LeftNode = new Markov::Node<unsigned char>('l');
00078                     Markov::Node<unsigned char>* RightNode = new Markov::Node<unsigned char>('r');
00079                     Markov::Edge<unsigned char>* e1 = new Markov::Edge<unsigned char>(LeftNode,
      RightNode);
00080
00081                     Markov::Edge<unsigned char>* e2 = new Markov::Edge<unsigned char>;
00082                     e2->SetLeftEdge(LeftNode);
00083                     e2->SetRightEdge(RightNode);
00084
00085                     Assert::IsTrue(e1->LeftNode() == e2->LeftNode());
00086                     Assert::IsTrue(e1->RightNode() == e2->RightNode());
00087                     delete LeftNode;
00088                     delete RightNode;
00089                     delete e1;
00090                     delete e2;
00091                 }
00092
00095                 TEST_METHOD(negative_adjust) {
00096                     Markov::Node<unsigned char>* LeftNode = new Markov::Node<unsigned char>('l');
00097                     Markov::Node<unsigned char>* RightNode = new Markov::Node<unsigned char>('r');
00098                     Markov::Edge<unsigned char>* e = new Markov::Edge<unsigned char>(LeftNode,
      RightNode);
00099                     e->AdjustEdge(15);
00100                    Assert::AreEqual(15ull, e->EdgeWeight());
00101                    e->AdjustEdge(-15);
00102                    Assert::AreEqual(0ull, e->EdgeWeight());
00103                    delete LeftNode;
00104                    delete RightNode;
00105                    delete e;
00106                }
00107            };
```

References TEST_CLASS().

Here is the call graph for this function:



7.18.2.2 TEST_CLASS() [2/3]

```
Testing::MVP::MarkovModel::TEST_CLASS (
            Model  )
```

Test class for minimal viable Model.

test model constructor for starter node

test import

test export

test random walk

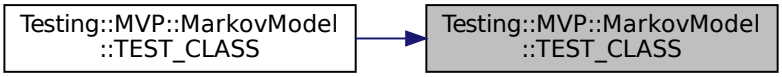
Definition at line 347 of file UnitTests.cpp.

```
00348            {
00349            public:
00352                TEST_METHOD(model_constructor) {
00353                    Markov::Model<unsigned char> m;
00354                    Assert::AreEqual((unsigned char)'\0', m.StarterNode()->NodeValue());
00355                }
00356
00359                TEST_METHOD(import_filename) {
00360                    Markov::Model<unsigned char> m;
00361                    Assert::IsTrue(m.Import("../MarkovPasswords/Models/2gram.mdl"));
00362                }
00363
00366                TEST_METHOD(export_filename) {
00367                    Markov::Model<unsigned char> m;
00368                    Assert::IsTrue(m.Export("../MarkovPasswords/Models/testcase.mdl"));
```

```
00369                 }
00370
00373                 TEST_METHOD(random_walk) {
00374                     unsigned char* res = new unsigned char[12 + 5];
00375                     Markov::Random::Marsaglia MarsagliaRandomEngine;
00376                     Markov::Model<unsigned char> m;
00377                     Assert::IsTrue(m.Import("../Models/finished2.mdl"));
00378                     Assert::IsNotNull(m.RandomWalk(&MarsagliaRandomEngine,1,12,res));
00379                 }
00380             };
```

References TEST_CLASS().
Referenced by TEST_CLASS().
Here is the call graph for this function:

Here is the caller graph for this function:



7.18.2.3 TEST_CLASS() [3/3]

```
Testing::MVP::MarkovModel::TEST_CLASS (
            Node  )
```

Test class for minimal viable Node.
test default constructor
test custom constructor with unsigned char
test link function
test link function
test RandomNext with low values
test RandomNext with 32 bit high values
random next on a node with no follow-ups
random next on a node with no follow-ups
test updateEdges
test updateEdges
test FindVertice
test FindVertice
test FindVertice
Definition at line 111 of file UnitTests.cpp.

```
00112             {
00113             public:
00114
00117                 TEST_METHOD(default_constructor) {
```

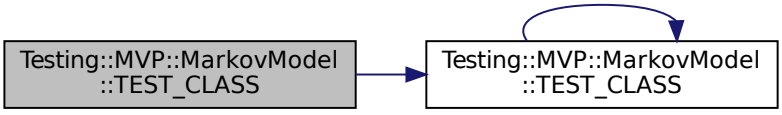
```
00118                   Markov::Node<unsigned char>* n = new Markov::Node<unsigned char>();
00119                   Assert::AreEqual((unsigned char)0, n->NodeValue());
00120                   delete n;
00121               }
00122
00125               TEST_METHOD(uchar_constructor) {
00126                   Markov::Node<unsigned char>* n = NULL;
00127                   unsigned char test_cases[] = { 'c', 0x00, 0xff, -32 };
00128                   for (unsigned char tcase : test_cases) {
00129                       n = new Markov::Node<unsigned char>(tcase);
00130                       Assert::AreEqual(tcase, n->NodeValue());
00131                       delete n;
00132                   }
00133               }
00134
00137               TEST_METHOD(link_left) {
00138                   Markov::Node<unsigned char>* LeftNode = new Markov::Node<unsigned char>('l');
00139                   Markov::Node<unsigned char>* RightNode = new Markov::Node<unsigned char>('r');
00140
00141                   Markov::Edge<unsigned char>* e = LeftNode->Link(RightNode);
00142                   delete LeftNode;
00143                   delete RightNode;
00144                   delete e;
00145               }
00146
00149               TEST_METHOD(link_right) {
00150                   Markov::Node<unsigned char>* LeftNode = new Markov::Node<unsigned char>('l');
00151                   Markov::Node<unsigned char>* RightNode = new Markov::Node<unsigned char>('r');
00152
00153                   Markov::Edge<unsigned char>* e = new Markov::Edge<unsigned char>(NULL, RightNode);
00154                   LeftNode->Link(e);
00155                   Assert::IsTrue(LeftNode == e->LeftNode());
00156                   Assert::IsTrue(RightNode == e->RightNode());
00157                   delete LeftNode;
00158                   delete RightNode;
00159                   delete e;
00160               }
00161
00164               TEST_METHOD(rand_next_low) {
00165                   Markov::Random::Marsaglia MarsagliaRandomEngine;
00166                   Markov::Node<unsigned char>* src = new Markov::Node<unsigned char>('a');
00167                   Markov::Node<unsigned char>* target1 = new Markov::Node<unsigned char>('b');
00168                   Markov::Edge<unsigned char>* e = src->Link(target1);
00169                   e->AdjustEdge(15);
00170                   Markov::Node<unsigned char>* res = src->RandomNext(&MarsagliaRandomEngine);
00171                   Assert::IsTrue(res == target1);
00172                   delete src;
00173                   delete target1;
00174                   delete e;
00175
00176               }
00177
00180               TEST_METHOD(rand_next_u32) {
00181                   Markov::Random::Marsaglia MarsagliaRandomEngine;
00182                   Markov::Node<unsigned char>* src = new Markov::Node<unsigned char>('a');
00183                   Markov::Node<unsigned char>* target1 = new Markov::Node<unsigned char>('b');
00184                   Markov::Edge<unsigned char>* e = src->Link(target1);
00185                   e->AdjustEdge(1 << 31);
00186                   Markov::Node<unsigned char>* res = src->RandomNext(&MarsagliaRandomEngine);
00187                   Assert::IsTrue(res == target1);
00188                   delete src;
00189                   delete target1;
00190                   delete e;
00191
00192               }
00193
00196               TEST_METHOD(rand_next_choice_1) {
00197                   Markov::Random::Marsaglia MarsagliaRandomEngine;
00198                   Markov::Node<unsigned char>* src = new Markov::Node<unsigned char>('a');
00199                   Markov::Node<unsigned char>* target1 = new Markov::Node<unsigned char>('b');
00200                   Markov::Node<unsigned char>* target2 = new Markov::Node<unsigned char>('c');
00201                   Markov::Edge<unsigned char>* e1 = src->Link(target1);
00202                   Markov::Edge<unsigned char>* e2 = src->Link(target2);
00203                   e1->AdjustEdge(1);
00204                   e2->AdjustEdge((unsigned long)(1ull << 31));
00205                   Markov::Node<unsigned char>* res = src->RandomNext(&MarsagliaRandomEngine);
00206                   Assert::IsNotNull(res);
00207                   Assert::IsTrue(res == target2);
00208                   delete src;
00209                   delete target1;
00210                   delete e1;
00211                   delete e2;
00212               }
00213
00216               TEST_METHOD(rand_next_choice_2) {
00217                   Markov::Random::Marsaglia MarsagliaRandomEngine;
00218                   Markov::Node<unsigned char>* src = new Markov::Node<unsigned char>('a');
```

```
00219                   Markov::Node<unsigned char>* target1 = new Markov::Node<unsigned char>('b');
00220                   Markov::Node<unsigned char>* target2 = new Markov::Node<unsigned char>('c');
00221                   Markov::Edge<unsigned char>* e1 = src->Link(target1);
00222                   Markov::Edge<unsigned char>* e2 = src->Link(target2);
00223                   e2->AdjustEdge(1);
00224                   e1->AdjustEdge((unsigned long)(1ull << 31));
00225                   Markov::Node<unsigned char>* res = src->RandomNext(&MarsagliaRandomEngine);
00226                   Assert::IsNotNull(res);
00227                   Assert::IsTrue(res == target1);
00228                   delete src;
00229                   delete target1;
00230                   delete e1;
00231                   delete e2;
00232               }
00233
00234
00237               TEST_METHOD(update_edges_count) {
00238
00239                   Markov::Node<unsigned char>* src = new Markov::Node<unsigned char>('a');
00240                   Markov::Node<unsigned char>* target1 = new Markov::Node<unsigned char>('b');
00241                   Markov::Node<unsigned char>* target2 = new Markov::Node<unsigned char>('c');
00242                   Markov::Edge<unsigned char>* e1 = new Markov::Edge<unsigned char>(src, target1);
00243                   Markov::Edge<unsigned char>* e2 = new Markov::Edge<unsigned char>(src, target2);
00244                   e1->AdjustEdge(25);
00245                   src->UpdateEdges(e1);
00246                   e2->AdjustEdge(30);
00247                   src->UpdateEdges(e2);
00248
00249                   Assert::AreEqual((size_t)2, src->Edges()->size());
00250
00251                   delete src;
00252                   delete target1;
00253                   delete e1;
00254                   delete e2;
00255
00256               }
00257
00260               TEST_METHOD(update_edges_total) {
00261
00262                   Markov::Node<unsigned char>* src = new Markov::Node<unsigned char>('a');
00263                   Markov::Node<unsigned char>* target1 = new Markov::Node<unsigned char>('b');
00264                   Markov::Edge<unsigned char>* e1 = new Markov::Edge<unsigned char>(src, target1);
00265                   Markov::Edge<unsigned char>* e2 = new Markov::Edge<unsigned char>(src, target1);
00266                   e1->AdjustEdge(25);
00267                   src->UpdateEdges(e1);
00268                   e2->AdjustEdge(30);
00269                   src->UpdateEdges(e2);
00270
00271                   //Assert::AreEqual(55ull, src->TotalEdgeWeights());
00272
00273                   delete src;
00274                   delete target1;
00275                   delete e1;
00276                   delete e2;
00277               }
00278
00279
00282               TEST_METHOD(find_vertice) {
00283
00284                   Markov::Node<unsigned char>* src = new Markov::Node<unsigned char>('a');
00285                   Markov::Node<unsigned char>* target1 = new Markov::Node<unsigned char>('b');
00286                   Markov::Node<unsigned char>* target2 = new Markov::Node<unsigned char>('c');
00287                   Markov::Edge<unsigned char>* res = NULL;
00288                   src->Link(target1);
00289                   src->Link(target2);
00290
00291
00292                   res = src->FindEdge('b');
00293                   Assert::IsNotNull(res);
00294                   Assert::AreEqual((unsigned char)'b', res->TraverseNode()->NodeValue());
00295                   res = src->FindEdge('c');
00296                   Assert::IsNotNull(res);
00297                   Assert::AreEqual((unsigned char)'c', res->TraverseNode()->NodeValue());
00298
00299                   delete src;
00300                   delete target1;
00301                   delete target2;
00302
00303
00304               }
00305
00306
00309               TEST_METHOD(find_vertice_without_any) {
00310
00311                   auto _invalid_next = [] {
00312                       Markov::Node<unsigned char>* src = new Markov::Node<unsigned char>('a');
00313                       Markov::Edge<unsigned char>* res = NULL;
```

```
00314
00315                         res = src->FindEdge('b');
00316                         Assert::IsNull(res);
00317
00318                         delete src;
00319                     };
00320
00321                     //Assert::ExpectException<std::logic_error>(_invalid_next);
00322                 }
00323
00326                 TEST_METHOD(find_vertice_nonexistent) {
00327
00328                     Markov::Node<unsigned char>* src = new Markov::Node<unsigned char>('a');
00329                     Markov::Node<unsigned char>* target1 = new Markov::Node<unsigned char>('b');
00330                     Markov::Node<unsigned char>* target2 = new Markov::Node<unsigned char>('c');
00331                     Markov::Edge<unsigned char>* res = NULL;
00332                     src->Link(target1);
00333                     src->Link(target2);
00334
00335                     res = src->FindEdge('D');
00336                     Assert::IsNull(res);
00337
00338                     delete src;
00339                     delete target1;
00340                     delete target2;
00341
00342                 }
00343             };
```

References TEST_CLASS().

Here is the call graph for this function:



7.19 Testing::MVP::MarkovPasswords Namespace Reference

Testing namespace for MVP MarkovPasswords.

Functions

- TEST_CLASS (ArgParser)

  Test Class for Argparse class.

7.19.1 Detailed Description

Testing namespace for MVP MarkovPasswords.

7.19.2 Function Documentation

7.19.2.1 TEST_CLASS()

```
Testing::MVP::MarkovPasswords::TEST_CLASS (
            ArgParser  )
```

Test Class for Argparse class.

test basic generate

test basic generate reordered params

test basic generate param longnames

test basic generate
test basic train
test basic generate
Definition at line 389 of file UnitTests.cpp.

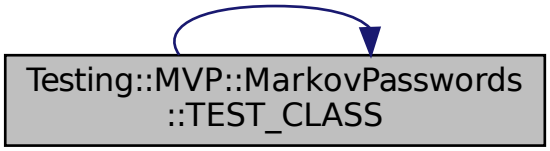
```
00390             {
00391             public:
00394                 TEST_METHOD(generate_basic) {
00395                     int argc = 8;
00396                     char *argv[] = {"markov.exe", "generate", "-if", "model.mdl", "-of",
      "passwords.txt", "-n", "100"};
00397
00398                     /*ProgramOptions *p = Argparse::parse(argc, argv);
00399                     Assert::IsNotNull(p);
00400
00401                     Assert::AreEqual(p->bImport, true);
00402                     Assert::AreEqual(p->bExport, false);
00403                     Assert::AreEqual(p->importname, "model.mdl");
00404                     Assert::AreEqual(p->outputfilename, "passwords.txt");
00405                     Assert::AreEqual(p->generateN, 100); */
00406
00407                 }
00408
00411                 TEST_METHOD(generate_basic_reorder) {
00412                     int argc = 8;
00413                     char *argv[] = { "markov.exe", "generate", "-n", "100", "-if", "model.mdl", "-of",
      "passwords.txt" };
00414
00415                     /*ProgramOptions* p = Argparse::parse(argc, argv);
00416                     Assert::IsNotNull(p);
00417
00418                     Assert::AreEqual(p->bImport, true);
00419                     Assert::AreEqual(p->bExport, false);
00420                     Assert::AreEqual(p->importname, "model.mdl");
00421                     Assert::AreEqual(p->outputfilename, "passwords.txt");
00422                     Assert::AreEqual(p->generateN, 100);*/
00423                 }
00424
00427                 TEST_METHOD(generate_basic_longname) {
00428                     int argc = 8;
00429                     char *argv[] = { "markov.exe", "generate", "-n", "100", "--inputfilename",
      "model.mdl", "--outputfilename", "passwords.txt" };
00430
00431                     /*ProgramOptions* p = Argparse::parse(argc, argv);
00432                     Assert::IsNotNull(p);
00433
00434                     Assert::AreEqual(p->bImport, true);
00435                     Assert::AreEqual(p->bExport, false);
00436                     Assert::AreEqual(p->importname, "model.mdl");
00437                     Assert::AreEqual(p->outputfilename, "passwords.txt");
00438                     Assert::AreEqual(p->generateN, 100); */
00439                 }
00440
00443                 TEST_METHOD(generate_fail_badmethod) {
00444                     int argc = 8;
00445                     char *argv[] = { "markov.exe", "junk", "-n", "100", "--inputfilename",
      "model.mdl", "--outputfilename", "passwords.txt" };
00446
00447                     /*ProgramOptions* p = Argparse::parse(argc, argv);
00448                     Assert::IsNull(p); */
00449                 }
00450
00453                 TEST_METHOD(train_basic) {
00454                     int argc = 4;
00455                     char *argv[] = { "markov.exe", "train", "-ef", "model.mdl" };
00456
00457                     /*ProgramOptions* p = Argparse::parse(argc, argv);
00458                     Assert::IsNotNull(p);
00459
00460                     Assert::AreEqual(p->bImport, false);
00461                     Assert::AreEqual(p->bExport, true);
00462                     Assert::AreEqual(p->exportname, "model.mdl"); */
00463
00464                 }
00465
00468                 TEST_METHOD(train_basic_longname) {
00469                     int argc = 4;
00470                     char *argv[] = { "markov.exe", "train", "--exportfilename", "model.mdl" };
00471
00472                     /*ProgramOptions* p = Argparse::parse(argc, argv);
00473                     Assert::IsNotNull(p);
00474
00475                     Assert::AreEqual(p->bImport, false);
00476                     Assert::AreEqual(p->bExport, true);
00477                     Assert::AreEqual(p->exportname, "model.mdl"); */
00478                 }
00479
```

```
00480
00481
00482             };
```

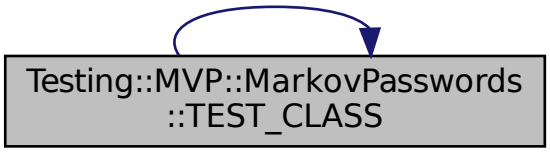
References TEST_CLASS().

Referenced by TEST_CLASS().

Here is the call graph for this function:



Here is the caller graph for this function:



Chapter 8

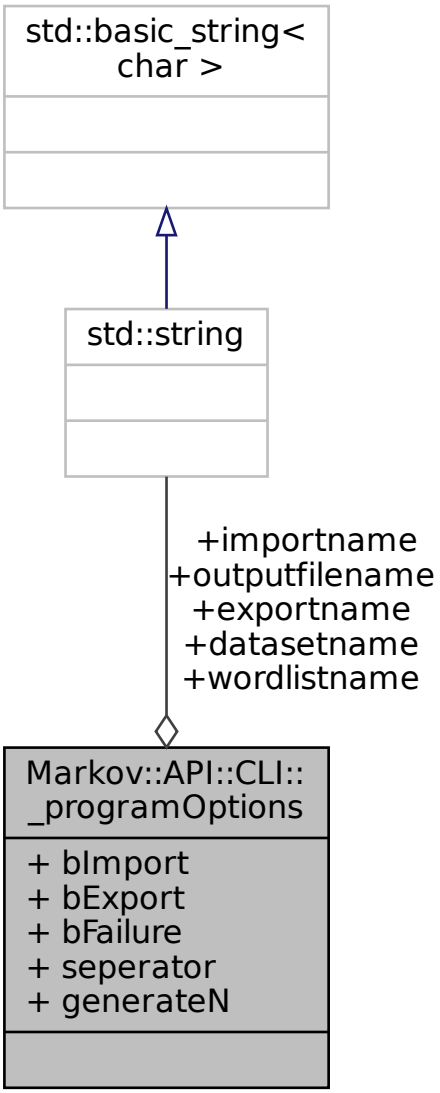
Class Documentation

8.1 Markov::API::CLI::_programOptions Struct Reference

Structure to hold parsed cli arguements.

`#include <argparse.h>`

Collaboration diagram for Markov::API::CLI::_programOptions:



Public Attributes

- bool bImport

  Import flag to validate import

- bool bExport

  Export flag to validate export

- bool bFailure

  Failure flag to validate succesfull running

- char seperator

Seperator character to use with training data. (character between occurence and value)"

- std::string importname

  Import name of our model.
- std::string exportname

  Import name of our given wordlist

- std::string wordlistname

  Import name of our given wordlist

- std::string outputfilename

  Output name of our generated password list

- std::string datasetname

  The name of the given dataset

- int generateN

  Number of passwords to be generated

8.1.1 Detailed Description

Structure to hold parsed cli arguements.
Definition at line 17 of file argparse.h.

8.1.2 Member Data Documentation

8.1.2.1 bExport

`bool Markov::API::CLI::_programOptions::bExport`
Export flag to validate export

Definition at line 26 of file argparse.h.
Referenced by Markov::API::CLI::Argparse::Argparse(), and Markov::API::CLI::Argparse::setProgramOptions().

8.1.2.2 bFailure

`bool Markov::API::CLI::_programOptions::bFailure`
Failure flag to validate succesfull running

Definition at line 31 of file argparse.h.
Referenced by Markov::API::CLI::Argparse::Argparse(), and Markov::API::CLI::Argparse::setProgramOptions().

8.1.2.3 bImport

`bool Markov::API::CLI::_programOptions::bImport`
Import flag to validate import

Definition at line 21 of file argparse.h.
Referenced by Markov::API::CLI::Argparse::Argparse(), and Markov::API::CLI::Argparse::setProgramOptions().

8.1.2.4 datasetname

`std::string Markov::API::CLI::_programOptions::datasetname`
The name of the given dataset

Definition at line 61 of file argparse.h.
Referenced by Markov::API::CLI::Argparse::Argparse(), and Markov::API::CLI::Argparse::setProgramOptions().

8.1.2.5 exportname

`std::string Markov::API::CLI::_programOptions::exportname`
Import name of our given wordlist

Definition at line 46 of file argparse.h.
Referenced by Markov::API::CLI::Argparse::setProgramOptions().

8.1.2.6 generateN

`int Markov::API::CLI::_programOptions::generateN`
Number of passwords to be generated

Definition at line 66 of file argparse.h.
Referenced by Markov::API::CLI::Argparse::Argparse(), and Markov::API::CLI::Argparse::setProgramOptions().

8.1.2.7 importname

`std::string Markov::API::CLI::_programOptions::importname`
Import name of our model.
Definition at line 41 of file argparse.h.
Referenced by Markov::API::CLI::Argparse::Argparse(), and Markov::API::CLI::Argparse::setProgramOptions().

8.1.2.8 outputfilename

`std::string Markov::API::CLI::_programOptions::outputfilename`
Output name of our generated password list

Definition at line 56 of file argparse.h.
Referenced by Markov::API::CLI::Argparse::Argparse(), and Markov::API::CLI::Argparse::setProgramOptions().

8.1.2.9 seperator

`char Markov::API::CLI::_programOptions::seperator`
Seperator character to use with training data. (character between occurence and value)"

Definition at line 36 of file argparse.h.
Referenced by Markov::API::CLI::Argparse::setProgramOptions().

8.1.2.10 wordlistname

`std::string Markov::API::CLI::_programOptions::wordlistname`
Import name of our given wordlist

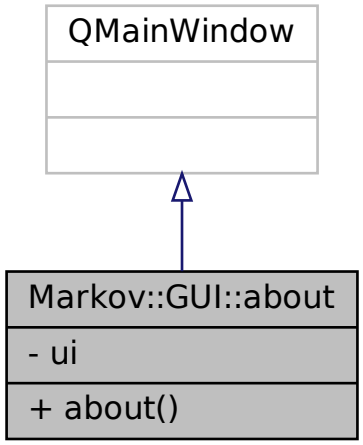
Definition at line 51 of file argparse.h.
Referenced by Markov::API::CLI::Argparse::Argparse().

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

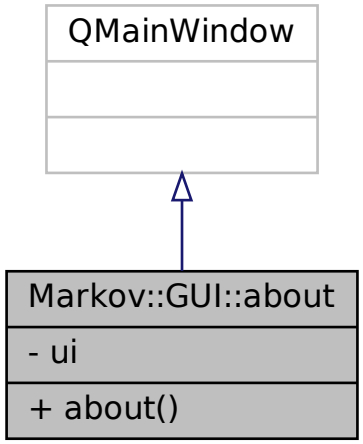
- argparse.h

8.2 Markov::GUI::about Class Reference

QT Class for about page.
`#include <about.h>`
Inheritance diagram for Markov::GUI::about:



Collaboration diagram for Markov::GUI::about:



Public Member Functions

- about (QWidget *parent=Q_NULLPTR)

Private Attributes

- Ui::main ui

8.2.1 Detailed Description

QT Class for about page.
Definition at line 12 of file about.h.

8.2.2 Constructor & Destructor Documentation

8.2.2.1 about()

```
Markov::GUI::about::about (
            QWidget * parent = Q_NULLPTR )
```

8.2.3 Member Data Documentation

8.2.3.1 ui

```
Ui::  main Markov::GUI::about::ui  [private]
```

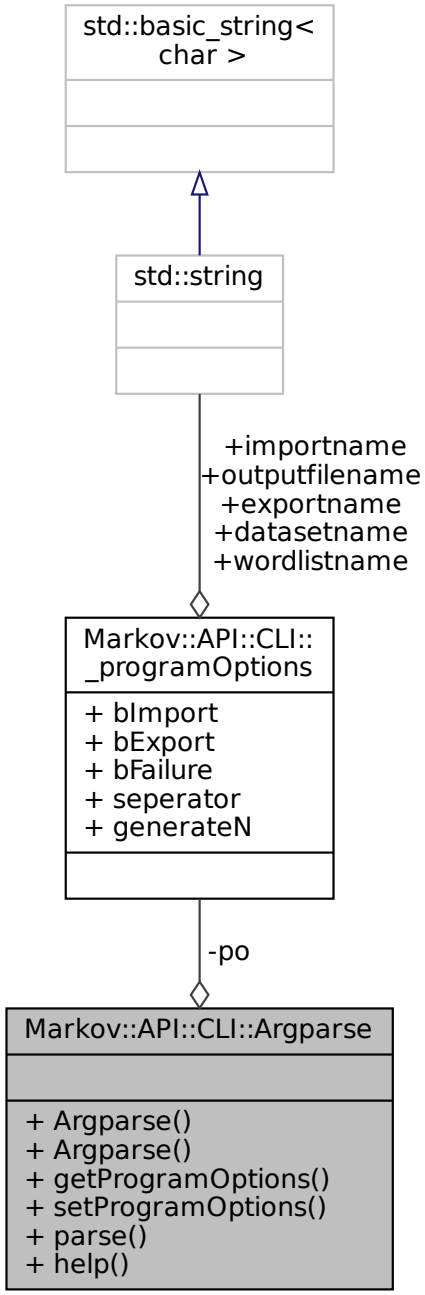
Definition at line 18 of file about.h.
The documentation for this class was generated from the following file:

- about.h

8.3 Markov::API::CLI::Argparse Class Reference

Parse command line arguements.
`#include <argparse.h>`

Collaboration diagram for Markov::API::CLI::Argparse:



Public Member Functions

- Argparse ()
- Argparse (int argc, char **argv)

  Parse command line arguements.
- Markov::API::CLI::ProgramOptions getProgramOptions (void)

  Getter for command line options.

- void setProgramOptions (bool i, bool e, bool bf, char s, std::string iName, std::string exName, std::string oName, std::string dName, int n)

  Initialize program options structure.

Static Public Member Functions

- static Markov::API::CLI::ProgramOptions * parse (int argc, char **argv)

  parse cli commands and return

- static void help ()

  Print help string.

Private Attributes

- Markov::API::CLI::ProgramOptions po

  ProgramOptions structure object.

8.3.1 Detailed Description

Parse command line arguements.
Definition at line 73 of file argparse.h.

8.3.2 Constructor & Destructor Documentation

8.3.2.1 Argparse() [1/2]

```
Markov::API::CLI::Argparse::Argparse ( )
```

8.3.2.2 Argparse() [2/2]

```
Markov::API::CLI::Argparse::Argparse (
            int argc,
            char ** argv )  [inline]
```

Parse command line arguements.
Parses command line arguements to populate ProgramOptions structure.

Parameters

argc	Number of command line arguements
argv	Array of command line parameters

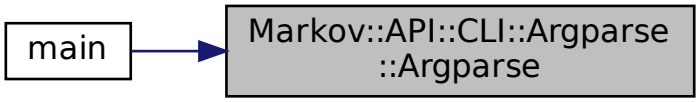
Definition at line 85 of file argparse.h.

```
00085                                       {
00086
00087             /*bool bImp;
00088             bool bExp;
00089             bool bFail;
00090             char sprt;
00091             std::string imports;
00092             std::string exports;
00093             std::string outputs;
00094             std::string datasets;
00095             int generateN;
00096             */
00097             opt::options_description desc("Options");
00098
00099
00100             desc.add_options()
00101                 ("generate", "Generate strings with given parameters")
00102                 ("train", "Train model with given parameters")
```

```
00103                 ("combine", "Combine")
00104                 ("import", opt::value<std::string>(), "Import model file")
00105                 ("output", opt::value<std::string>(), "Output model file. This model will be exported
      when done. Will be ignored for generation mode")
00106                 ("dataset", opt::value<std::string>(), "Dataset file to read input from training. Will
      be ignored for generation mode")
00107                 ("seperator", opt::value<char>(), "Seperator character to use with training data.
      (character between occurence and value)")
00108                 ("wordlist", opt::value<std::string>(), "Wordlist file path to export generation
      results to. Will be ignored for training mode")
00109                 ("count", opt::value<int>(), "Number of lines to generate. Ignored in training mode")
00110                 ("verbosity", "Output verbosity")
00111                 ("help", "Option definitions");
00112
00113                 opt::variables_map vm;
00114
00115                 opt::store(opt::parse_command_line(argc, argv, desc), vm);
00116
00117                 opt::notify(vm);
00118
00119                 //std::cout << desc << std::endl;
00120                 if (vm.count("help")) {
00121                 std::cout << desc << std::endl;
00122                 }
00123
00124                 if (vm.count("output") == 0) this->po.outputfilename = "NULL";
00125                 else if (vm.count("output") == 1) {
00126                     this->po.outputfilename = vm["output"].as<std::string>();
00127                     this->po.bExport = true;
00128                 }
00129                 else {
00130                     this->po.bFailure = true;
00131                     std::cout << "UNIDENTIFIED INPUT" << std::endl;
00132                     std::cout << desc << std::endl;
00133                 }
00134
00135
00136                 if (vm.count("dataset") == 0) this->po.datasetname = "NULL";
00137                 else if (vm.count("dataset") == 1) {
00138                     this->po.datasetname = vm["dataset"].as<std::string>();
00139                 }
00140                 else {
00141                     this->po.bFailure = true;
00142                     std::cout << "UNIDENTIFIED INPUT" << std::endl;
00143                     std::cout << desc << std::endl;
00144                 }
00145
00146
00147                 if (vm.count("wordlist") == 0) this->po.wordlistname = "NULL";
00148                 else if (vm.count("wordlist") == 1) {
00149                     this->po.wordlistname = vm["wordlist"].as<std::string>();
00150                 }
00151                 else {
00152                     this->po.bFailure = true;
00153                     std::cout << "UNIDENTIFIED INPUT" << std::endl;
00154                     std::cout << desc << std::endl;
00155                 }
00156
00157                 if (vm.count("import") == 0) this->po.importname = "NULL";
00158                 else if (vm.count("import") == 1) {
00159                     this->po.importname = vm["import"].as<std::string>();
00160                     this->po.bImport = true;
00161                 }
00162                 else {
00163                     this->po.bFailure = true;
00164                     std::cout << "UNIDENTIFIED INPUT" << std::endl;
00165                     std::cout << desc << std::endl;
00166                 }
00167
00168
00169                 if (vm.count("count") == 0) this->po.generateN = 0;
00170                 else if (vm.count("count") == 1) {
00171                     this->po.generateN = vm["count"].as<int>();
00172                 }
00173                 else {
00174                     this->po.bFailure = true;
00175                     std::cout << "UNIDENTIFIED INPUT" << std::endl;
00176                     std::cout << desc << std::endl;
00177                 }
00178
00179                 /*std::cout << vm["output"].as<std::string>() << std::endl;
00180                 std::cout << vm["dataset"].as<std::string>() << std::endl;
00181                 std::cout << vm["wordlist"].as<std::string>() << std::endl;
00182                 std::cout << vm["output"].as<std::string>() << std::endl;
00183                 std::cout << vm["count"].as<int>() << std::endl;*/
00184
00185
```

```
00186                 //else if (vm.count("train")) std::cout « "train oldu" « std::endl;
00187         }
```

References Markov::API::CLI::_programOptions::bExport, Markov::API::CLI::_programOptions::bFailure, Markov::API::CLI::_program
Markov::API::CLI::_programOptions::datasetname, Markov::API::CLI::_programOptions::generateN, Markov::API::CLI::_programOpti
Markov::API::CLI::_programOptions::outputfilename, po, and Markov::API::CLI::_programOptions::wordlistname.
Referenced by main().
Here is the caller graph for this function:



8.3.3 Member Function Documentation

8.3.3.1 getProgramOptions()

```
Markov::API::CLI::ProgramOptions Markov::API::CLI::Argparse::getProgramOptions (
            void  )  [inline]
```

Getter for command line options.
Getter for ProgramOptions populated by the arguement parser

Returns

ProgramOptions structure.

Definition at line 194 of file argparse.h.

```
00194                                                   {
00195             return this->po;
00196         }
```

References po.

8.3.3.2 help()

```
void Markov::API::CLI::Argparse::help ( )  [static]
```

Print help string.
Definition at line 8 of file argparse.cpp.

```
00008                                {
00009     std::cout «
00010     "Markov Passwords - Help\n"
00011     "Options:\n"
00012     "   \n"
00013     "   -of --outputfilename\n"
00014     "       Filename to output the generation results\n"
00015     "   -ef --exportfilename\n"
00016     "       filename to export built model to\n"
00017     "   -if --importfilename\n"
00018     "       filename to import model from\n"
00019     "   -n (generate count)\n"
00020     "       Number of lines to generate\n"
00021     "   \n"
00022     "Usage: \n"
00023     "   markov.exe -if empty_model.mdl -ef model.mdl\n"
00024     "       import empty_model.mdl and train it with data from stdin. When done, output the model to
      model.mdl\n"
00025     "\n"
00026     "   markov.exe -if empty_model.mdl -n 15000 -of wordlist.txt\n"
00027     "       import empty_model.mdl and generate 15000 words to wordlist.txt\n"
00028
```

```
00029       « std::endl;
00030 }
```

8.3.3.3 parse()

```
Markov::API::CLI::ProgramOptions * Markov::API::CLI::Argparse::parse (
            int argc,
            char ** argv )  [static]
```

parse cli commands and return

Parameters

argc	- Program arguement count
argv	- Program arguement values array

Returns

ProgramOptions structure.

Definition at line 4 of file argparse.cpp.

```
00004 { return 0; }
```

8.3.3.4 setProgramOptions()

```
void Markov::API::CLI::Argparse::setProgramOptions (
            bool i,
            bool e,
            bool bf,
            char s,
            std::string iName,
            std::string exName,
            std::string oName,
            std::string dName,
            int n )  [inline]
```

Initialize program options structure.

Parameters

i	boolean, true if import operation is flagged
e	boolean, true if export operation is flagged
bf	boolean, true if there is something wrong with the command line parameters
s	seperator character for the import function
iName	import filename
exName	export filename
oName	output filename
dName	corpus filename
n	number of passwords to be generated

Definition at line 211 of file argparse.h.

```
00211
                                        {
00212             this->po.bImport = i;
00213             this->po.bExport = e;
00214             this->po.seperator = s;
00215             this->po.bFailure = bf;
00216             this->po.generateN = n;
00217             this->po.importname = iName;
```

```
00218             this->po.exportname = exName;
00219             this->po.outputfilename = oName;
00220             this->po.datasetname = dName;
00221 
00222             /*strcpy_s(this->po.importname,256,iName);
00223             strcpy_s(this->po.exportname,256,exName);
00224             strcpy_s(this->po.outputfilename,256,oName);
00225             strcpy_s(this->po.datasetname,256,dName);*/
00226 
00227         }
```

References Markov::API::CLI::_programOptions::bExport, Markov::API::CLI::_programOptions::bFailure, Markov::API::CLI::_program
Markov::API::CLI::_programOptions::datasetname, Markov::API::CLI::_programOptions::exportname, Markov::API::CLI::_programOp
Markov::API::CLI::_programOptions::importname, Markov::API::CLI::_programOptions::outputfilename, po, and
Markov::API::CLI::_programOptions::seperator.

8.3.4 Member Data Documentation

8.3.4.1 po

Markov::API::CLI::ProgramOptions Markov::API::CLI::Argparse::po [private]

ProgramOptions structure object.

Definition at line 246 of file argparse.h.

Referenced by Argparse(), getProgramOptions(), and setProgramOptions().

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

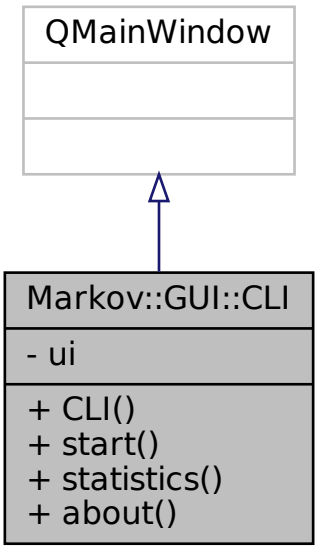
- argparse.h
- argparse.cpp

8.4 Markov::GUI::CLI Class Reference

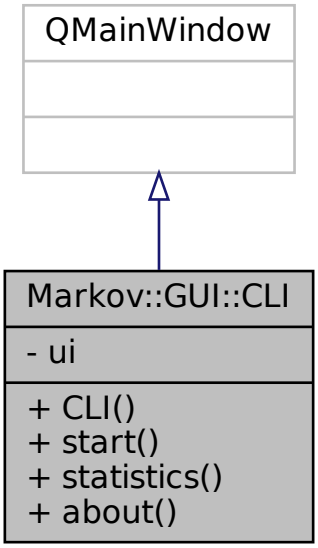
QT CLI Class.

`#include <CLI.h>`

Inheritance diagram for Markov::GUI::CLI:



Collaboration diagram for Markov::GUI::CLI:



Public Slots

- void start ()
- void statistics ()
- void about ()

Public Member Functions

- CLI (QWidget *parent=Q_NULLPTR)

Private Attributes

- Ui::CLI ui

8.4.1 Detailed Description

QT CLI Class.
Definition at line 8 of file CLI.h.

8.4.2 Constructor & Destructor Documentation

8.4.2.1 CLI()

```
Markov::GUI::CLI::CLI (
            QWidget * parent = Q_NULLPTR )
```

8.4.3 Member Function Documentation

8.4.3.1 about

```
void Markov::GUI::CLI::about ( )  [slot]
```

8.4.3.2 start

```
void Markov::GUI::CLI::start ( )  [slot]
```

Referenced by main().
Here is the caller graph for this function:

8.4.3.3 statistics

```
void Markov::GUI::CLI::statistics ( )  [slot]
```

8.4.4 Member Data Documentation

8.4.4.1 ui

```
Ui::CLI Markov::GUI::CLI::ui  [private]
```

Definition at line 14 of file CLI.h.
The documentation for this class was generated from the following file:

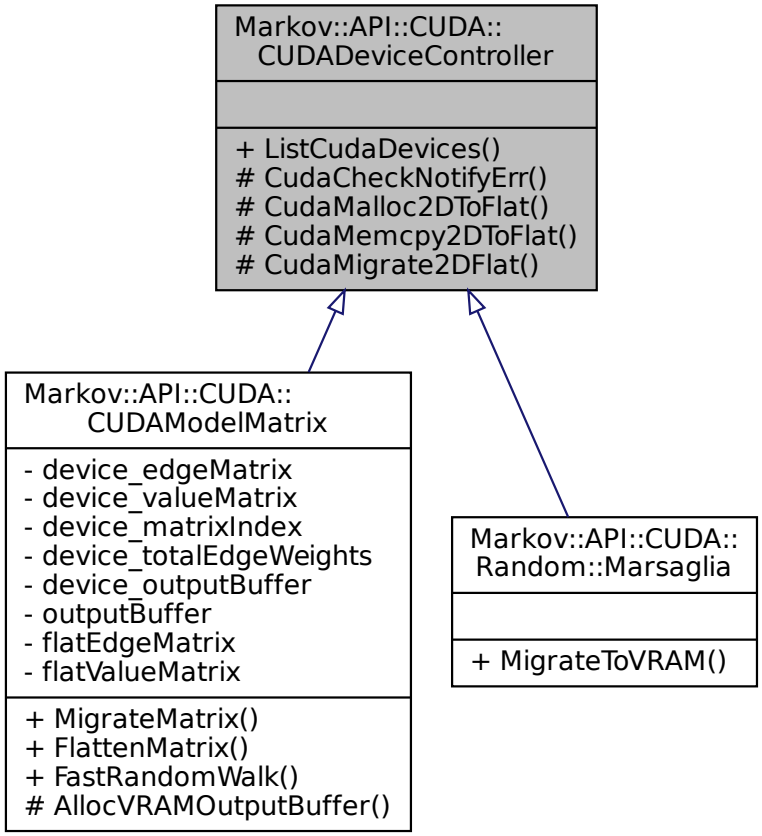
- CLI.h

8.5 Markov::API::CUDA::CUDADeviceController Class Reference

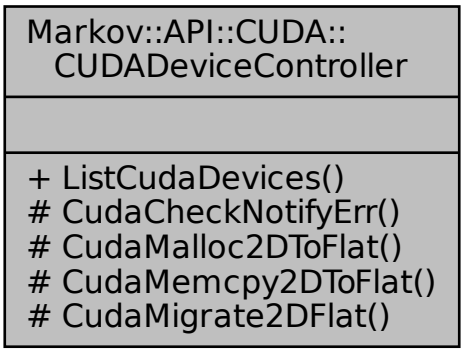
Controller class for CUDA device.

```
#include <cudaDeviceController.h>
```

Inheritance diagram for Markov::API::CUDA::CUDADeviceController:



Collaboration diagram for Markov::API::CUDA::CUDADeviceController:



Static Public Member Functions

- static __host__ void ListCudaDevices ()

  List CUDA devices in the system.

Static Protected Member Functions

- static __host__ int CudaCheckNotifyErr (cudaError_t _status, const char *msg, bool bExit=true)

  Check results of the last operation on GPU.
- template<typename T >
  static __host__ cudaError_t CudaMalloc2DToFlat (T **dst, int row, int col)

  Malloc a 2D array in device space.
- template<typename T >
  static __host__ cudaError_t CudaMemcpy2DToFlat (T *dst, T **src, int row, int col)

  Memcpy a 2D array in device space after flattening.
- template<typename T >
  static __host__ cudaError_t CudaMigrate2DFlat (T **dst, T **src, int row, int col)

  Both malloc and memcpy a 2D array into device VRAM.

8.5.1 Detailed Description

Controller class for CUDA device.
This implementation only supports Nvidia devices.
Definition at line 16 of file cudaDeviceController.h.

8.5.2 Member Function Documentation

8.5.2.1 CudaCheckNotifyErr()

```
static __host__ int Markov::API::CUDA::CUDADeviceController::CudaCheckNotifyErr (
            cudaError_t _status,
            const char * msg,
            bool bExit = true )  [static], [protected]
```

Check results of the last operation on GPU.
Check the status returned from cudaMalloc/cudaMemcpy to find failures.
If a failure occurs, its assumed beyond redemption, and exited.

Parameters

_status	Cuda error status to check
msg	Message to print in case of a failure

Returns

0 if successful, 1 if failure. **Example output:**

```
char *da, a = "test";
cudastatus = cudaMalloc((char **)&da, 5*sizeof(char*));
CudaCheckNotifyErr(cudastatus, "Failed to allocate VRAM for *da.\n");
```

8.5.2.2 CudaMalloc2DToFlat()

```
template<typename T >
static __host__ cudaError_t Markov::API::CUDA::CUDADeviceController::CudaMalloc2DToFlat (
            T ** dst,
```

```
            int row,
            int col )  [inline], [static], [protected]
```

Malloc a 2D array in device space.
This function will allocate enough space on VRAM for flattened 2D array.

Parameters

dst	destination pointer
row	row size of the 2d array
col	column size of the 2d array

Returns

cudaError_t status of the cudaMalloc operation

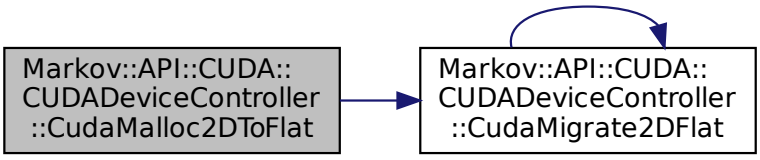
Example output:

```
cudaError_t cudastatus;
char* dst;
cudastatus = CudaMalloc2DToFlat<char>(&dst, 5, 15);
if(cudastatus!=cudaSuccess){
    CudaCheckNotifyErr(cudastatus, "  CudaMalloc2DToFlat Failed.", false);
}
```

Definition at line 73 of file cudaDeviceController.h.

```
00073                                                                       {
00074             cudaError_t cudastatus = cudaMalloc((T **)dst, row*col*sizeof(T));
00075             CudaCheckNotifyErr(cudastatus, "cudaMalloc Failed.", false);
00076             return cudastatus;
00077         }
```

References CudaMigrate2DFlat().
Here is the call graph for this function:



8.5.2.3 CudaMemcpy2DToFlat()

```
template<typename T >
static __host__ cudaError_t Markov::API::CUDA::CUDADeviceController::CudaMemcpy2DToFlat (
            T * dst,
            T ** src,
            int row,
            int col )  [inline], [static], [protected]
```

Memcpy a 2D array in device space after flattening.
Resulting buffer will not be true 2D array.

Parameters

dst	destination pointer
rc	source pointer
row	row size of the 2d array
col	column size of the 2d array

Returns

cudaError_t status of the cudaMalloc operation

Example output:

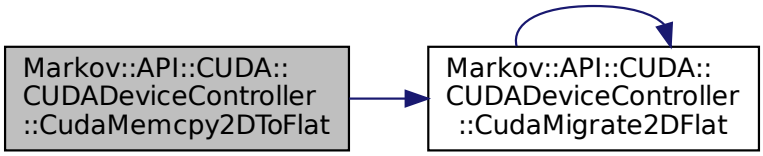
```
cudaError_t cudastatus;
char* dst;
cudastatus = CudaMalloc2DToFlat<char>(&dst, 5, 15);
CudaCheckNotifyErr(cudastatus, "  CudaMalloc2DToFlat Failed.", false);
cudastatus = CudaMemcpy2DToFlat<char>(*dst,src,15,15);
CudaCheckNotifyErr(cudastatus, "  CudaMemcpy2DToFlat Failed.", false);
```

Definition at line 101 of file cudaDeviceController.h.

```
00101                                                                                                {
00102             T* tempbuf = new T[row*col];
00103             for(int i=0;i<row;i++){
00104                 memcpy(&(tempbuf[row*i]), src[i], col);
00105             }
00106             return cudaMemcpy(dst, tempbuf, row*col*sizeof(T), cudaMemcpyHostToDevice);
00107
00108         }
```

References CudaMigrate2DFlat().

Here is the call graph for this function:



8.5.2.4 CudaMigrate2DFlat()

```
template<typename T >
static __host__ cudaError_t Markov::API::CUDA::CUDADeviceController::CudaMigrate2DFlat (
             T ** dst,
             T ** src,
             int row,
             int col )  [inline], [static], [protected]
```

Both malloc and memcpy a 2D array into device VRAM.

Resulting buffer will not be true 2D array.

Parameters

dst	destination pointer
rc	source pointer
row	row size of the 2d array
col	column size of the 2d array

Returns

cudaError_t status of the cudaMalloc operation

Example output:

```
cudaError_t cudastatus;
char* dst;
cudastatus = CudaMigrate2DFlat<long int>(
   &dst, this->valueMatrix, this->matrixSize, this->matrixSize);
CudaCheckNotifyErr(cudastatus, "    Cuda failed to initialize value matrix row.");
```

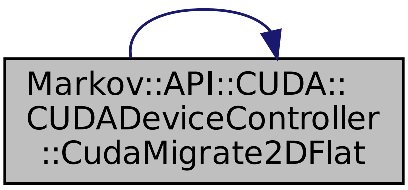
Definition at line 130 of file cudaDeviceController.h.

```
00130                                                                 {
00131         cudaError_t cudastatus;
00132         cudastatus = CudaMalloc2DToFlat<T>(dst, row, col);
00133         if(cudastatus!=cudaSuccess){
00134             CudaCheckNotifyErr(cudastatus, "  CudaMalloc2DToFlat Failed.", false);
00135             return cudastatus;
00136         }
00137         cudastatus = CudaMemcpy2DToFlat<T>(*dst,src,row,col);
00138         CudaCheckNotifyErr(cudastatus, "  CudaMemcpy2DToFlat Failed.", false);
00139         return cudastatus;
00140     }
```

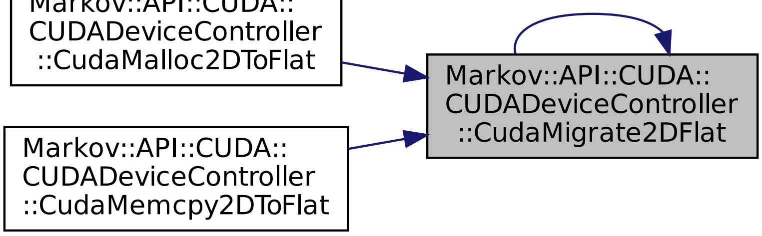
References CudaMigrate2DFlat().

Referenced by CudaMalloc2DToFlat(), CudaMemcpy2DToFlat(), and CudaMigrate2DFlat().

Here is the call graph for this function:



Here is the caller graph for this function:



8.5.2.5 ListCudaDevices()

```
static __host__ void Markov::API::CUDA::CUDADeviceController::ListCudaDevices ( )  [static]
```

List CUDA devices in the system.

This function will print details of every CUDA capable device in the system.

Example output:

```
Device Number: 0
Device name: GeForce RTX 2070
Memory Clock Rate (KHz): 7001000
Memory Bus Width (bits): 256
Peak Memory Bandwidth (GB/s): 448.064
Max Linear Threads: 1024
```

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

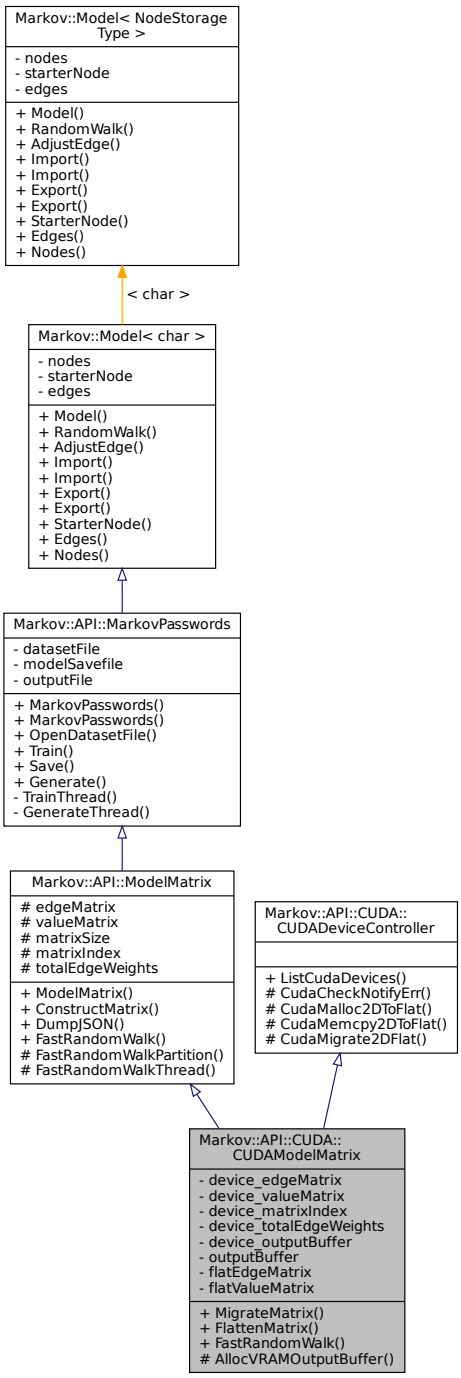
- cudaDeviceController.h

8.6 Markov::API::CUDA::CUDAModelMatrix Class Reference

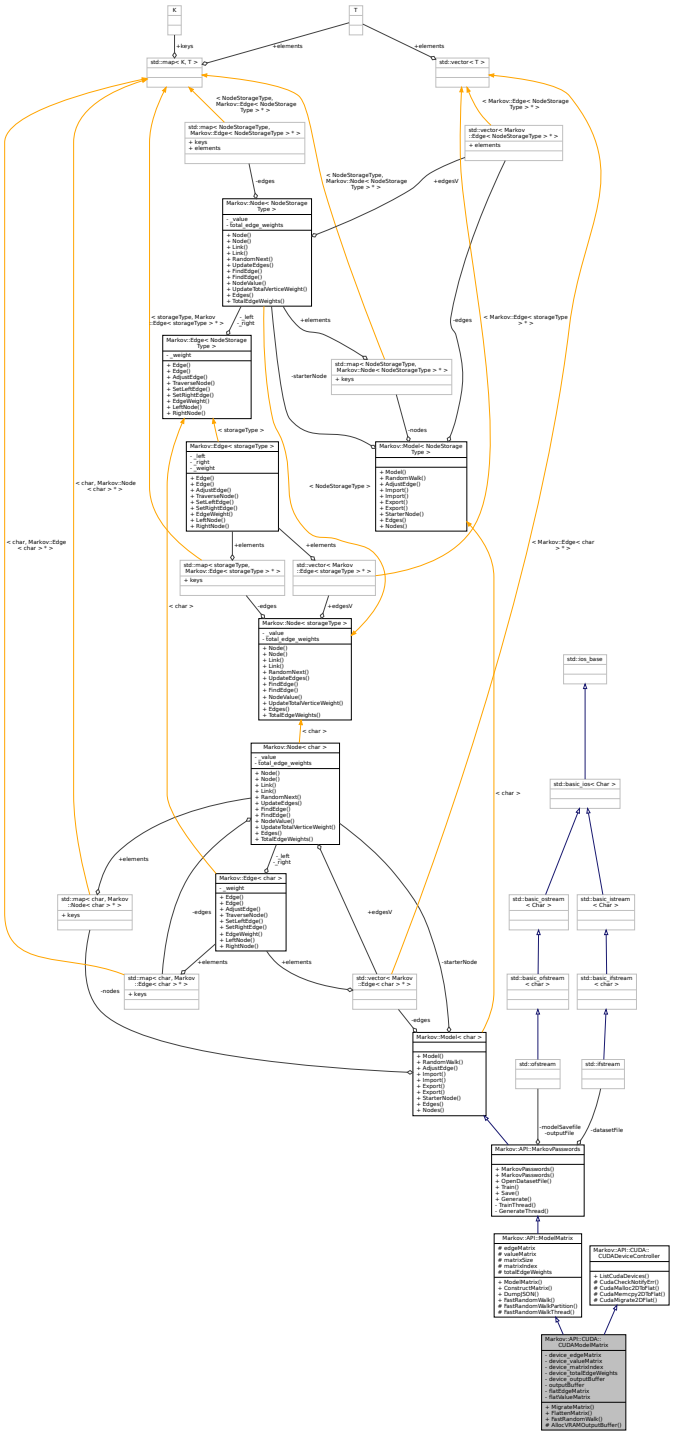
Extension of Markov::API::ModelMatrix which is modified to run on GPU devices.

`#include <cudaModelMatrix.h>`

Inheritance diagram for Markov::API::CUDA::CUDAModelMatrix:



Collaboration diagram for Markov::API::CUDA::CUDAModelMatrix:



Public Member Functions

- __host__ void MigrateMatrix ()

  Migrate the class members to the VRAM.
- __host__ void FlattenMatrix ()

  Flatten migrated matrix from 2d to 1d.
- __host__ void FastRandomWalk (unsigned long int n, const char ∗wordlistFileName, int minLen, int maxLen, bool bFileIO)

Random walk on the Matrix-reduced Markov::Model.

- void ConstructMatrix ()

  Construct the related Matrix data for the model.
- void DumpJSON ()

  Debug function to dump the model to a JSON file.
- void FastRandomWalk (unsigned long int n, const char *wordlistFileName, int minLen=6, int maxLen=12, int threads=20, bool bFileIO=true)

  Random walk on the Matrix-reduced Markov::Model.
- std::ifstream * OpenDatasetFile (const char *filename)

  Open dataset file and return the ifstream pointer.
- void Train (const char *datasetFileName, char delimiter, int threads)

  Train the model with the dataset file.
- std::ofstream * Save (const char *filename)

  Export model to file.
- void Generate (unsigned long int n, const char *wordlistFileName, int minLen=6, int maxLen=12, int threads=20)

  Call Markov::Model::RandomWalk n times, and collect output.
- char * RandomWalk (Markov::Random::RandomEngine *randomEngine, int minSetting, int maxSetting, char *buffer)

  Do a random walk on this model.
- void AdjustEdge (const char *payload, long int occurrence)

  Adjust the model with a single string.
- bool Import (std::ifstream *)

  Import a file to construct the model.
- bool Import (const char *filename)

  Open a file to import with filename, and call bool Model::Import with std::ifstream.
- bool Export (std::ofstream *)

  Export a file of the model.
- bool Export (const char *filename)

  Open a file to export with filename, and call bool Model::Export with std::ofstream.
- Node< char > * StarterNode ()

  Return starter Node.
- std::vector< Edge< char > * > * Edges ()

  Return a vector of all the edges in the model.
- std::map< char, Node< char > * > * Nodes ()

  Return starter Node.

Static Public Member Functions

- static __host__ void ListCudaDevices ()

  List CUDA devices in the system.

Protected Member Functions

- __host__ char * AllocVRAMOutputBuffer (long int n, long int singleGenMaxLen, long int CUDAKernelGrid↩ Size, long int sizePerGrid)

  Allocate the output buffer for kernel operation.
- void FastRandomWalkPartition (std::mutex *mlock, std::ofstream *wordlist, unsigned long int n, int minLen, int maxLen, bool bFileIO, int threads)

  A single partition of FastRandomWalk event.
- void FastRandomWalkThread (std::mutex *mlock, std::ofstream *wordlist, unsigned long int n, int minLen, int maxLen, int id, bool bFileIO)

  A single thread of a single partition of FastRandomWalk.

Static Protected Member Functions

- static __host__ int CudaCheckNotifyErr (cudaError_t _status, const char *msg, bool bExit=true)

  Check results of the last operation on GPU.
- template<typename T >
  static __host__ cudaError_t CudaMalloc2DToFlat (T **dst, int row, int col)

  Malloc a 2D array in device space.
- template<typename T >
  static __host__ cudaError_t CudaMemcpy2DToFlat (T *dst, T **src, int row, int col)

  Memcpy a 2D array in device space after flattening.
- template<typename T >
  static __host__ cudaError_t CudaMigrate2DFlat (T **dst, T **src, int row, int col)

  Both malloc and memcpy a 2D array into device VRAM.

Protected Attributes

- char ** edgeMatrix

  2-D Character array for the edge Matrix (The characters of Nodes)
- long int ** valueMatrix

  2-d Integer array for the value Matrix (For the weights of Edges)
- int matrixSize

  to hold Matrix size
- char * matrixIndex

  to hold the Matrix index (To hold the orders of 2-D arrays')
- long int * totalEdgeWeights

  Array of the Total Edge Weights.

Private Member Functions

- void TrainThread (Markov::API::Concurrency::ThreadSharedListHandler *listhandler, char delimiter)

  A single thread invoked by the Train function.
- void GenerateThread (std::mutex *outputLock, unsigned long int n, std::ofstream *wordlist, int minLen, int maxLen)

  A single thread invoked by the Generate function.

Private Attributes

- char * device_edgeMatrix

  VRAM Address pointer of edge matrix (from modelMatrix.h)
- long int * device_valueMatrix

  VRAM Address pointer of value matrix (from modelMatrix.h)
- char * device_matrixIndex

  VRAM Address pointer of matrixIndex (from modelMatrix.h)
- long int * device_totalEdgeWeights

  VRAM Address pointer of total edge weights (from modelMatrix.h)
- char * device_outputBuffer

  RandomWalk results in device.
- char * outputBuffer

  RandomWalk results in host.
- char * flatEdgeMatrix

  Adding Edge matrix end-to-end and resize to 1-D array for better perfomance on traversing.
- long int * flatValueMatrix

  Adding Value matrix end-to-end and resize to 1-D array for better perfomance on traversing.

- std::ifstream * datasetFile
- std::ofstream * modelSavefile

  Dataset file input of our system

- std::ofstream * outputFile

  File to save model of our system

- std::map< char, Node< char > * > nodes

  Map LeftNode is the Nodes NodeValue Map RightNode is the node pointer.
- Node< char > * starterNode

  Starter Node of this model.
- std::vector< Edge< char > * > edges

  A list of all edges in this model.

8.6.1 Detailed Description

Extension of Markov::API::ModelMatrix which is modified to run on GPU devices.
This implementation only supports Nvidia devices.
Definition at line 11 of file cudaModelMatrix.h.

8.6.2 Member Function Documentation

8.6.2.1 AdjustEdge()

```
void Markov::Model< char >::AdjustEdge (
            const NodeStorageType * payload,
            long int occurrence )  [inherited]
```

Adjust the model with a single string.
Start from the starter node, and for each character, AdjustEdge the edge EdgeWeight from current node to the next, until NULL character is reached.
Then, update the edge EdgeWeight from current node, to the terminator node.
This function is used for training purposes, as it can be used for adjusting the model with each line of the corpus file.
Example Use: Create an empty model and train it with string: "testdata"

```
Markov::Model<char> model;
char test[] = "testdata";
model.AdjustEdge(test, 15);
```

Parameters

string	- String that is passed from the training, and will be used to AdjustEdge the model with
occurrence	- Occurrence of this string.

Definition at line 323 of file model.h.

```
00323                                                                                                {
00324     NodeStorageType p = payload[0];
00325     Markov::Node<NodeStorageType>* curnode = this->starterNode;
00326     Markov::Edge<NodeStorageType>* e;
00327     int i = 0;
00328
00329     if (p == 0) return;
00330     while (p != 0) {
00331         e = curnode->FindEdge(p);
00332         if (e == NULL) return;
00333         e->AdjustEdge(occurrence);
00334         curnode = e->RightNode();
00335         p = payload[++i];
00336     }
00337
00338     e = curnode->FindEdge('\xff');
00339     e->AdjustEdge(occurrence);
00340     return;
```

```
00341 }
```

8.6.2.2 AllocVRAMOutputBuffer()

```
__host__ char* Markov::API::CUDA::CUDAModelMatrix::AllocVRAMOutputBuffer (
            long int n,
            long int singleGenMaxLen,
            long int CUDAKernelGridSize,
            long int sizePerGrid )  [protected]
```

Allocate the output buffer for kernel operation.
TODO

Parameters

n	- Number of passwords to generate.
singleGenMaxLen	- maximum string length for a single generation
CUDAKernelGridSize	- Total number of grid members in CUDA kernel
sizePerGrid	- Size to allocate per grid member

Returns

pointer to the allocation on VRAM

8.6.2.3 ConstructMatrix()

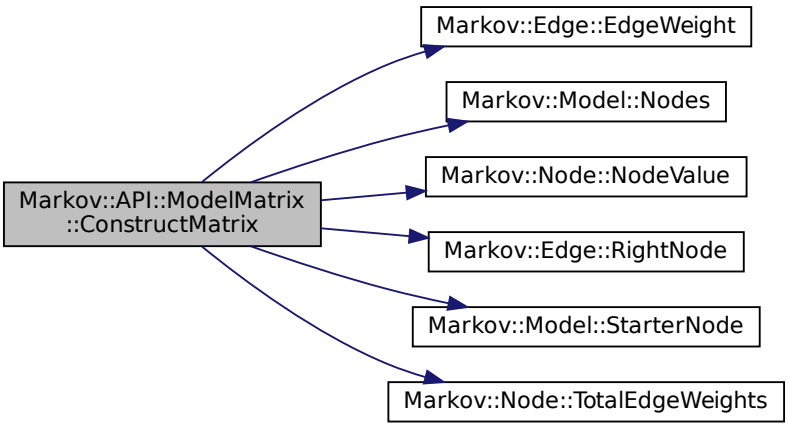
```
void Markov::API::ModelMatrix::ConstructMatrix ( )  [inherited]
```

Construct the related Matrix data for the model.
This operation can be used after importing/training to allocate and populate the matrix content.
this will initialize: char** edgeMatrix -> a 2D array of mapping left and right connections of each edge. long int **valueMatrix -> a 2D array representing the edge weights. int matrixSize -> Size of the matrix, aka total number of nodes. char* matrixIndex -> order of nodes in the model long int *totalEdgeWeights -> total edge weights of each Node.
Definition at line 11 of file modelMatrix.cpp.

```
00011                                           {
00012     this->matrixSize = this->StarterNode()->edgesV.size() + 2;
00013
00014     this->matrixIndex = new char[this->matrixSize];
00015     this->totalEdgeWeights = new long int[this->matrixSize];
00016
00017     this->edgeMatrix = new char*[this->matrixSize];
00018     for(int i=0;i<this->matrixSize;i++){
00019         this->edgeMatrix[i] = new char[this->matrixSize];
00020     }
00021     this->valueMatrix = new long int*[this->matrixSize];
00022     for(int i=0;i<this->matrixSize;i++){
00023         this->valueMatrix[i] = new long int[this->matrixSize];
00024     }
00025     std::map< char, Node< char > * > *nodes;
00026     nodes = this->Nodes();
00027     int i=0;
00028     for (auto const& [repr, node] : *nodes){
00029         if(repr!=0) this->matrixIndex[i] = repr;
00030         else this->matrixIndex[i] = 199;
00031         this->totalEdgeWeights[i] = node->TotalEdgeWeights();
00032         for(int j=0;j<this->matrixSize;j++){
00033             char val = node->NodeValue();
00034             if(val < 0){
00035                 for(int k=0;k<this->matrixSize;k++){
00036                     this->valueMatrix[i][k] = 0;
00037                     this->edgeMatrix[i][k] = 255;
00038                 }
00039                 break;
00040             }
00041             else if(node->NodeValue() == 0 && j>(this->matrixSize-3)){
00042                 this->valueMatrix[i][j] = 0;
00043                 this->edgeMatrix[i][j] = 255;
```

```
00044             }else if(j==(this->matrixSize-1)) {
00045                 this->valueMatrix[i][j] = 0;
00046                 this->edgeMatrix[i][j] = 255;
00047             }else{
00048                 this->valueMatrix[i][j] = node->edgesV[j]->EdgeWeight();
00049                 this->edgeMatrix[i][j]  = node->edgesV[j]->RightNode()->NodeValue();
00050             }
00051 
00052         }
00053         i++;
00054     }
00055 
00056     //this->DumpJSON();
00057 }
```

References Markov::API::ModelMatrix::edgeMatrix, Markov::Edge< NodeStorageType >::EdgeWeight(), Markov::API::ModelMatrix::n
Markov::API::ModelMatrix::matrixSize, Markov::Model< NodeStorageType >::Nodes(), Markov::Node< storageType >::NodeValue()
Markov::Edge< NodeStorageType >::RightNode(), Markov::Model< NodeStorageType >::StarterNode(), Markov::API::ModelMatrix:
Markov::Node< storageType >::TotalEdgeWeights(), and Markov::API::ModelMatrix::valueMatrix.
Referenced by main().
Here is the call graph for this function:



Here is the caller graph for this function:

8.6.2.4 CudaCheckNotifyErr()

```
static __host__ int Markov::API::CUDA::CUDADeviceController::CudaCheckNotifyErr (
            cudaError_t _status,
            const char * msg,
            bool bExit = true )  [static], [protected], [inherited]
```

Check results of the last operation on GPU.

Check the status returned from cudaMalloc/cudaMemcpy to find failures.
If a failure occurs, its assumed beyond redemption, and exited.

Parameters

_status	Cuda error status to check
msg	Message to print in case of a failure

Returns

0 if successful, 1 if failure. **Example output:**

```
char *da, a = "test";
cudastatus = cudaMalloc((char **)&da, 5*sizeof(char*));
CudaCheckNotifyErr(cudastatus, "Failed to allocate VRAM for *da.\n");
```

8.6.2.5 CudaMalloc2DToFlat()

```
template<typename T >
static __host__ cudaError_t Markov::API::CUDA::CUDADeviceController::CudaMalloc2DToFlat (
            T ** dst,
            int row,
            int col )  [inline], [static], [protected], [inherited]
```

Malloc a 2D array in device space.
This function will allocate enough space on VRAM for flattened 2D array.

Parameters

dst	destination pointer
row	row size of the 2d array
col	column size of the 2d array

Returns

cudaError_t status of the cudaMalloc operation

Example output:

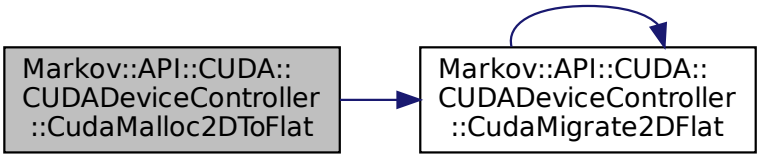
```
cudaError_t cudastatus;
char* dst;
cudastatus = CudaMalloc2DToFlat<char>(&dst, 5, 15);
if(cudastatus!=cudaSuccess){
    CudaCheckNotifyErr(cudastatus, "  CudaMalloc2DToFlat Failed.", false);
}
```

Definition at line 73 of file cudaDeviceController.h.

```
00073                                                                   {
00074             cudaError_t cudastatus = cudaMalloc((T **)dst, row*col*sizeof(T));
00075             CudaCheckNotifyErr(cudastatus, "cudaMalloc Failed.", false);
00076             return cudastatus;
00077         }
```

References Markov::API::CUDA::CUDADeviceController::CudaMigrate2DFlat().

Here is the call graph for this function:



8.6.2.6 CudaMemcpy2DToFlat()

```
template<typename T >
static __host__ cudaError_t Markov::API::CUDA::CUDADeviceController::CudaMemcpy2DToFlat (
            T * dst,
            T ** src,
            int row,
            int col )  [inline], [static], [protected], [inherited]
```

Memcpy a 2D array in device space after flattening.
Resulting buffer will not be true 2D array.

Parameters

dst	destination pointer
rc	source pointer
row	row size of the 2d array
col	column size of the 2d array

Returns

cudaError_t status of the cudaMalloc operation

Example output:

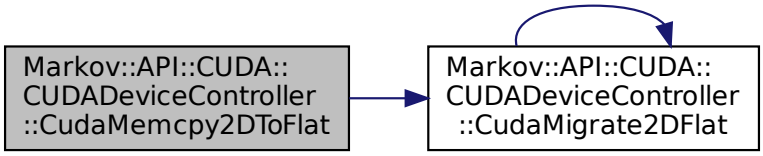
```
cudaError_t cudastatus;
char* dst;
cudastatus = CudaMalloc2DToFlat<char>(&dst, 5, 15);
CudaCheckNotifyErr(cudastatus, "  CudaMalloc2DToFlat Failed.", false);
cudastatus = CudaMemcpy2DToFlat<char>(*dst,src,15,15);
CudaCheckNotifyErr(cudastatus, "  CudaMemcpy2DToFlat Failed.", false);
```

Definition at line 101 of file cudaDeviceController.h.

```
00101                                                                                {
00102             T* tempbuf = new T[row*col];
00103             for(int i=0;i<row;i++){
00104                 memcpy(&(tempbuf[row*i]), src[i], col);
00105             }
00106             return cudaMemcpy(dst, tempbuf, row*col*sizeof(T), cudaMemcpyHostToDevice);
00107
00108         }
```

References Markov::API::CUDA::CUDADeviceController::CudaMigrate2DFlat().

Here is the call graph for this function:



8.6.2.7 CudaMigrate2DFlat()

```
template<typename T >
static __host__ cudaError_t Markov::API::CUDA::CUDADeviceController::CudaMigrate2DFlat (
            T ** dst,
            T ** src,
            int row,
            int col )  [inline], [static], [protected], [inherited]
```

Both malloc and memcpy a 2D array into device VRAM.
Resulting buffer will not be true 2D array.

Parameters

dst	destination pointer
rc	source pointer
row	row size of the 2d array
col	column size of the 2d array

Returns

cudaError_t status of the cudaMalloc operation

Example output:

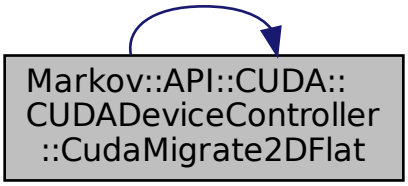
```
cudaError_t cudastatus;
char* dst;
cudastatus = CudaMigrate2DFlat<long int>(
   &dst, this->valueMatrix, this->matrixSize, this->matrixSize);
CudaCheckNotifyErr(cudastatus, "    Cuda failed to initialize value matrix row.");
```

Definition at line 130 of file cudaDeviceController.h.

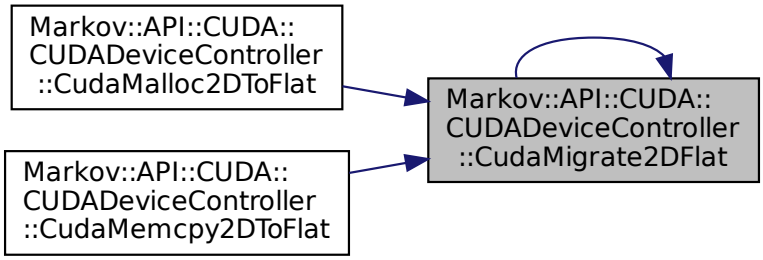
```
00130                                                                                {
00131             cudaError_t cudastatus;
00132             cudastatus = CudaMalloc2DToFlat<T>(dst, row, col);
00133             if(cudastatus!=cudaSuccess){
00134                 CudaCheckNotifyErr(cudastatus, "  CudaMalloc2DToFlat Failed.", false);
00135                 return cudastatus;
00136             }
00137             cudastatus = CudaMemcpy2DToFlat<T>(*dst,src,row,col);
00138             CudaCheckNotifyErr(cudastatus, "  CudaMemcpy2DToFlat Failed.", false);
00139             return cudastatus;
00140         }
```

References Markov::API::CUDA::CUDADeviceController::CudaMigrate2DFlat().
Referenced by Markov::API::CUDA::CUDADeviceController::CudaMalloc2DToFlat(), Markov::API::CUDA::CUDADeviceController::Cu
and Markov::API::CUDA::CUDADeviceController::CudaMigrate2DFlat().

Here is the call graph for this function:



Here is the caller graph for this function:



8.6.2.8 DumpJSON()

```
void Markov::API::ModelMatrix::DumpJSON ( )  [inherited]
```

Debug function to dump the model to a JSON file.
Might not work 100%. Not meant for production use.
Definition at line 60 of file modelMatrix.cpp.

```
00060                                {
00061
00062     std::cout « "{\n   \"index\": \"";
00063     for(int i=0;i<this->matrixSize;i++){
00064         if(this->matrixIndex[i]=='"') std::cout « "\\\"";
00065         else if(this->matrixIndex[i]=='\\') std::cout « "\\\\";
00066         else if(this->matrixIndex[i]==0) std::cout « "\\\\x00";
00067         else if(i==0) std::cout « "\\\\xff";
00068         else if(this->matrixIndex[i]=='\n') std::cout « "\\n";
00069         else std::cout « this->matrixIndex[i];
00070     }
00071     std::cout «
00072     "\",\n"
00073     "   \"edgemap\": {\n";
00074
00075     for(int i=0;i<this->matrixSize;i++){
00076         if(this->matrixIndex[i]=='"') std::cout « "      \"\\\"\": [";
00077         else if(this->matrixIndex[i]=='\\') std::cout « "      \"\\\\\": [";
00078         else if(this->matrixIndex[i]==0) std::cout « "      \"\\\\x00\": [";
00079         else if(this->matrixIndex[i]<0) std::cout « "      \"\\\\xff\": [";
00080         else std::cout « "      \"" « this->matrixIndex[i] « "\": [";
00081         for(int j=0;j<this->matrixSize;j++){
00082             if(this->edgeMatrix[i][j]=='"') std::cout « "\"\\\"\"";
00083             else if(this->edgeMatrix[i][j]=='\\') std::cout « "\"\\\\\"";
00084             else if(this->edgeMatrix[i][j]==0) std::cout « "\"\\\\x00\"";
00085             else if(this->edgeMatrix[i][j]<0) std::cout « "\"\\\\xff\"";
```

```
00086             else if(this->matrixIndex[i]=='\n') std::cout « "\"\\n\"";
00087             else std::cout « "\"" « this->edgeMatrix[i][j] « "\"";
00088             if(j!=this->matrixSize-1) std::cout « ", ";
00089         }
00090         std::cout « "],\n";
00091     }
00092     std::cout « "},\n";
00093
00094     std::cout « "\"   weightmap\": {\n";
00095     for(int i=0;i<this->matrixSize;i++){
00096         if(this->matrixIndex[i]=='"') std::cout « "      \"\\\"\": [";
00097         else if(this->matrixIndex[i]=='\\') std::cout « "      \"\\\\\": [";
00098         else if(this->matrixIndex[i]==0) std::cout « "      \"\\\\x00\": [";
00099         else if(this->matrixIndex[i]<0) std::cout « "      \"\\\\xff\": [";
00100         else std::cout « "      \"" « this->matrixIndex[i] « "\": [";
00101
00102         for(int j=0;j<this->matrixSize;j++){
00103             std::cout « this->valueMatrix[i][j];
00104             if(j!=this->matrixSize-1) std::cout « ", ";
00105         }
00106         std::cout « "],\n";
00107     }
00108     std::cout « "  }\n}\n";
00109 }
```

References Markov::API::ModelMatrix::edgeMatrix, Markov::API::ModelMatrix::matrixIndex, Markov::API::ModelMatrix::matrixSize, and Markov::API::ModelMatrix::valueMatrix.

8.6.2.9 Edges()

```
std::vector<Edge<char >*>* Markov::Model< char >::Edges ( )  [inline], [inherited]
```

Return a vector of all the edges in the model.

Returns

vector of edges

Definition at line 172 of file model.h.

```
00172 { return &edges;}
```

8.6.2.10 Export() [1/2]

```
bool Markov::Model< char >::Export (
            const char * filename )  [inherited]
```

Open a file to export with filename, and call bool Model::Export with std::ofstream.

Returns

True if successful, False for incomplete models or corrupt file formats

Example Use: Export file to filename

```
Markov::Model<char> model;
model.Export("test.mdl");
```

Definition at line 286 of file model.h.

```
00286                                                                {
00287     std::ofstream exportfile;
00288     exportfile.open(filename);
00289     return this->Export(&exportfile);
00290 }
```

8.6.2.11 Export() [2/2]

```
bool Markov::Model< char >::Export (
            std::ofstream * f )  [inherited]
```

Export a file of the model.
File contains a list of edges. Format is: Left_repr;EdgeWeight;right_repr. For more information on the format, check out the project wiki or github readme.
Iterate over this vertices, and their edges, and write them to file.

Returns

True if successful, False for incomplete models.

Example Use: Export file to ofstream

```
Markov::Model<char> model;
std::ofstream file("test.mdl");
model.Export(&file);
```

Definition at line 274 of file model.h.

```
00274                                                   {
00275     Markov::Edge<NodeStorageType>* e;
00276     for (std::vector<int>::size_type i = 0; i != this->edges.size(); i++) {
00277         e = this->edges[i];
00278         //std::cout « e->LeftNode()->NodeValue() « "," « e->EdgeWeight() « "," «
      e->RightNode()->NodeValue() « "\n";
00279         *f « e->LeftNode()->NodeValue() « "," « e->EdgeWeight() « "," « e->RightNode()->NodeValue() «
      "\n";
00280     }
00281
00282     return true;
00283 }
```

8.6.2.12 FastRandomWalk() [1/2]

```
__host__ void Markov::API::CUDA::CUDAModelMatrix::FastRandomWalk (
            unsigned long int n,
            const char * wordlistFileName,
            int minLen,
            int maxLen,
            bool bFileIO )
```

Random walk on the Matrix-reduced Markov::Model.
TODO

Parameters

n	- Number of passwords to generate.
wordlistFileName	- Filename to write to
minLen	- Minimum password length to generate
maxLen	- Maximum password length to generate
threads	- number of OS threads to spawn
bFileIO	- If false, filename will be ignored and will output to stdout.

```
Markov::API::ModelMatrix mp;
mp.Import("models/finished.mdl");
mp.FastRandomWalk(50000000,"./wordlist.txt",6,12,25, true);
```

8.6.2.13 FastRandomWalk() [2/2]

```
void Markov::API::ModelMatrix::FastRandomWalk (
            unsigned long int n,
            const char * wordlistFileName,
            int minLen = 6,
            int maxLen = 12,
            int threads = 20,
            bool bFileIO = true )  [inherited]
```

Random walk on the Matrix-reduced Markov::Model.
This has an O(N) Memory complexity. To limit the maximum usage, requests with n>50M are partitioned using Markov::API::ModelMatrix::FastRandomWalkPartition.
If n>50M, threads are going to be synced, files are going to be flushed, and buffers will be reallocated every 50M generations. This comes at a minor performance penalty.
While it has the same functionality, this operation reduces Markov::API::MarkovPasswords::Generate runtime by %96.5
This function has deprecated Markov::API::MarkovPasswords::Generate, and will eventually replace it.

Parameters

n	- Number of passwords to generate.
wordlistFileName	- Filename to write to
minLen	- Minimum password length to generate
maxLen	- Maximum password length to generate
threads	- number of OS threads to spawn
bFileIO	- If false, filename will be ignored and will output to stdout.

```
Markov::API::ModelMatrix mp;
mp.Import("models/finished.mdl");
mp.FastRandomWalk(50000000,"./wordlist.txt",6,12,25, true);
```

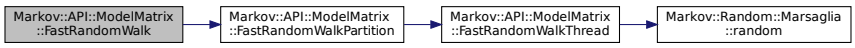
Definition at line 163 of file modelMatrix.cpp.

```
00163                                                {
00164
00165
00166     std::ofstream wordlist;
00167     if(bFileIO)
00168         wordlist.open(wordlistFileName);
00169
00170     std::mutex mlock;
00171     if(n<=50000000ull) return this->FastRandomWalkPartition(&mlock, &wordlist, n, minLen, maxLen,
      bFileIO, threads);
00172     else{
00173         int numberOfPartitions = n/50000000ull;
00174         for(int i=0;i<numberOfPartitions;i++)
00175             this->FastRandomWalkPartition(&mlock, &wordlist, 50000000ull, minLen, maxLen, bFileIO,
      threads);
00176     }
00177
00178
00179 }
```

References Markov::API::ModelMatrix::FastRandomWalkPartition().

Referenced by main().

Here is the call graph for this function:



Here is the caller graph for this function:

8.6.2.14 FastRandomWalkPartition()

```
void Markov::API::ModelMatrix::FastRandomWalkPartition (
            std::mutex * mlock,
            std::ofstream * wordlist,
            unsigned long int n,
            int minLen,
```

```
            int maxLen,
            bool bFileIO,
            int threads )  [protected], [inherited]
```

A single partition of FastRandomWalk event.
Since FastRandomWalk has to allocate its output buffer before operation starts and writes data in chunks, large n parameters would lead to huge memory allocations. **Without Partitioning:**

- 50M results 12 characters max -> 550 Mb Memory allocation
- 5B results 12 characters max -> 55 Gb Memory allocation
- 50B results 12 characters max -> 550GB Memory allocation

Instead, FastRandomWalk is partitioned per 50M generations to limit the top memory need.

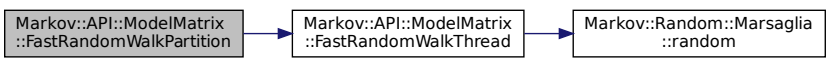
Parameters

mlock	- mutex lock to distribute to child threads
wordlist	- Reference to the wordlist file to write to
n	- Number of passwords to generate.
wordlistFileName	- Filename to write to
minLen	- Minimum password length to generate
maxLen	- Maximum password length to generate
threads	- number of OS threads to spawn
bFileIO	- If false, filename will be ignored and will output to stdout.

Definition at line 182 of file modelMatrix.cpp.

```
00182                                                                   {
00183
00184     int iterationsPerThread = n/threads;
00185     int iterationsPerThreadCarryOver = n%threads;
00186
00187     std::vector<std::thread*> threadsV;
00188
00189     int id = 0;
00190     for(int i=0;i<threads;i++){
00191         threadsV.push_back(new std::thread(&Markov::API::ModelMatrix::FastRandomWalkThread, this,
      mlock, wordlist, iterationsPerThread, minLen, maxLen, id, bFileIO));
00192         id++;
00193     }
00194
00195     threadsV.push_back(new std::thread(&Markov::API::ModelMatrix::FastRandomWalkThread, this, mlock,
     wordlist, iterationsPerThreadCarryOver, minLen, maxLen, id, bFileIO));
00196
00197     for(int i=0;i<threads;i++){
00198         threadsV[i]->join();
00199     }
00200 }
```

References Markov::API::ModelMatrix::FastRandomWalkThread().
Referenced by Markov::API::ModelMatrix::FastRandomWalk().
Here is the call graph for this function:



Here is the caller graph for this function:

8.6.2.15 FastRandomWalkThread()

```
void Markov::API::ModelMatrix::FastRandomWalkThread (
            std::mutex * mlock,
            std::ofstream * wordlist,
            unsigned long int n,
            int minLen,
            int maxLen,
            int id,
            bool bFileIO )  [protected], [inherited]
```

A single thread of a single partition of FastRandomWalk.
A FastRandomWalkPartition will initiate as many of this function as requested.
This function contains the bulk of the generation algorithm.

Parameters

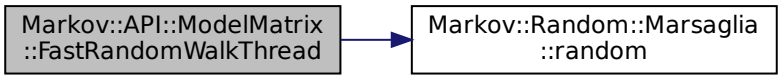
mlock	- mutex lock to distribute to child threads
wordlist	- Reference to the wordlist file to write to
n	- Number of passwords to generate.
wordlistFileName	- Filename to write to
minLen	- Minimum password length to generate
maxLen	- Maximum password length to generate
id	- **DEPRECATED** Thread id - No longer used
bFileIO	- If false, filename will be ignored and will output to stdout.

Definition at line 112 of file modelMatrix.cpp.

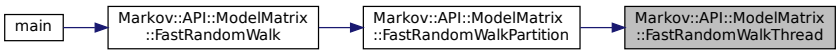
```
00112                                                       {
00113     if(n==0) return;
00114
00115     Markov::Random::Marsaglia MarsagliaRandomEngine;
00116     char* e;
00117     char *res = new char[maxLen*n];
00118     int index = 0;
00119     char next;
00120     int len=0;
00121     long int selection;
00122     char cur;
00123     long int bufferctr = 0;
00124     for (int i = 0; i < n; i++) {
00125         cur=199;
00126         len=0;
00127         while (true) {
00128             e = strchr(this->matrixIndex, cur);
00129             index = e - this->matrixIndex;
00130             selection = MarsagliaRandomEngine.random() % this->totalEdgeWeights[index];
00131             for(int j=0;j<this->matrixSize;j++){
00132                 selection -= this->valueMatrix[index][j];
00133                 if (selection < 0){
00134                     next = this->edgeMatrix[index][j];
00135                     break;
00136                 }
00137             }
00138
00139             if (len >= maxLen)  break;
```

```
00140             else if ((next < 0) && (len < minLen)) continue;
00141             else if (next < 0) break;
00142             cur = next;
00143             res[bufferctr + len++] = cur;
00144         }
00145         res[bufferctr + len++] = '\n';
00146         bufferctr+=len;
00147 
00148     }
00149     if(bFileIO){
00150         mlock->lock();
00151         *wordlist « res;
00152         mlock->unlock();
00153     }else{
00154         mlock->lock();
00155         std::cout « res;
00156         mlock->unlock();
00157     }
00158     delete res;
00159 
00160 }
```

References Markov::API::ModelMatrix::edgeMatrix, Markov::API::ModelMatrix::matrixIndex, Markov::API::ModelMatrix::matrixSize, Markov::Random::Marsaglia::random(), Markov::API::ModelMatrix::totalEdgeWeights, and Markov::API::ModelMatrix::valueMatrix.
Referenced by Markov::API::ModelMatrix::FastRandomWalkPartition().
Here is the call graph for this function:



Here is the caller graph for this function:



8.6.2.16 FlattenMatrix()

```
__host__ void Markov::API::CUDA::CUDAModelMatrix::FlattenMatrix ( )
```

Flatten migrated matrix from 2d to 1d.

8.6.2.17 Generate()

```
void Markov::API::MarkovPasswords::Generate (
            unsigned long int n,
            const char * wordlistFileName,
            int minLen = 6,
            int maxLen = 12,
            int threads = 20 )  [inherited]
```

Call Markov::Model::RandomWalk n times, and collect output.
Generate from model and write results to a file. a much more performance-optimized method. FastRandomWalk will reduce the runtime by %96.5 on average.

Deprecated See Markov::API::MatrixModel::FastRandomWalk for more information.

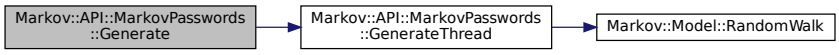
Parameters

n	- Number of passwords to generate.
wordlistFileName	- Filename to write to
minLen	- Minimum password length to generate
maxLen	- Maximum password length to generate
threads	- number of OS threads to spawn

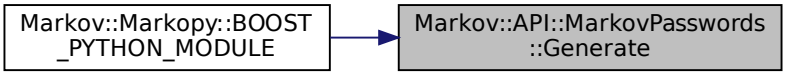
Definition at line 110 of file markovPasswords.cpp.

```
00110                                {
00111     char* res;
00112     char print[100];
00113     std::ofstream wordlist;
00114     wordlist.open(wordlistFileName);
00115     std::mutex mlock;
00116     int iterationsPerThread = n/threads;
00117     int iterationsCarryOver = n%threads;
00118     std::vector<std::thread*> threadsV;
00119     for(int i=0;i<threads;i++){
00120         threadsV.push_back(new std::thread(&Markov::API::MarkovPasswords::GenerateThread, this,
      &mlock, iterationsPerThread, &wordlist, minLen, maxLen));
00121     }
00122
00123     for(int i=0;i<threads;i++){
00124         threadsV[i]->join();
00125         delete threadsV[i];
00126     }
00127
00128     this->GenerateThread(&mlock, iterationsCarryOver, &wordlist, minLen, maxLen);
00129
00130 }
```

References Markov::API::MarkovPasswords::GenerateThread().
Referenced by Markov::Markopy::BOOST_PYTHON_MODULE().
Here is the call graph for this function:



Here is the caller graph for this function:



8.6.2.18 GenerateThread()

```
void Markov::API::MarkovPasswords::GenerateThread (
             std::mutex * outputLock,
             unsigned long int n,
             std::ofstream * wordlist,
             int minLen,
             int maxLen )  [private], [inherited]
```

A single thread invoked by the Generate function.

DEPRECATED: See Markov::API::MatrixModel::FastRandomWalkThread for more information. This has been replaced with a much more performance-optimized method. FastRandomWalk will reduce the runtime by %96.5 on average.

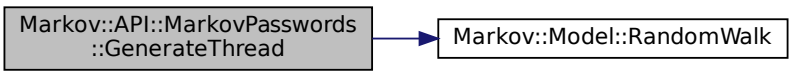
Parameters

outputLock	- shared mutex lock to lock during output operation. Prevents race condition on write.
n	number of lines to be generated by this thread
wordlist	wordlistfile
minLen	- Minimum password length to generate
maxLen	- Maximum password length to generate

Definition at line 132 of file markovPasswords.cpp.

```
00132                                                {
00133     char* res = new char[maxLen+5];
00134     if(n==0) return;
00135
00136     Markov::Random::Marsaglia MarsagliaRandomEngine;
00137     for (int i = 0; i < n; i++) {
00138         this->RandomWalk(&MarsagliaRandomEngine, minLen, maxLen, res);
00139         outputLock->lock();
00140         *wordlist « res « "\n";
00141         outputLock->unlock();
00142     }
00143 }
```

References Markov::Model< NodeStorageType >::RandomWalk().
Referenced by Markov::API::MarkovPasswords::Generate().
Here is the call graph for this function:



Here is the caller graph for this function:

8.6.2.19 Import() [1/2]

```
bool Markov::Model< char >::Import (
            const char * filename )  [inherited]
```

Open a file to import with filename, and call bool Model::Import with std::ifstream.

Returns

True if successful, False for incomplete models or corrupt file formats

Example Use: Import a file with filename

```
Markov::Model<char> model;
model.Import("test.mdl");
```

Definition at line 266 of file model.h.

```
00266                                                {
```

```
00267     std::ifstream importfile;
00268     importfile.open(filename);
00269     return this->Import(&importfile);
00270
00271 }
```

8.6.2.20 Import() [2/2]

```
bool Markov::Model< char >::Import (
            std::ifstream * f )  [inherited]
```

Import a file to construct the model.
File contains a list of edges. For more info on the file format, check out the wiki and github readme pages. Format is: Left_repr;EdgeWeight;right_repr
Iterate over this list, and construct nodes and edges accordingly.

Returns

True if successful, False for incomplete models or corrupt file formats

Example Use: Import a file from ifstream

```
Markov::Model<char> model;
std::ifstream file("test.mdl");
model.Import(&file);
```

Definition at line 207 of file model.h.

```
00207                                                   {
00208     std::string cell;
00209
00210     char src;
00211     char target;
00212     long int oc;
00213
00214     while (std::getline(*f, cell)) {
00215         //std::cout « "cell: " « cell « std::endl;
00216         src = cell[0];
00217         target = cell[cell.length() - 1];
00218         char* j;
00219         oc = std::strtol(cell.substr(2, cell.length() - 2).c_str(),&j,10);
00220         //std::cout « oc « "\n";
00221         Markov::Node<NodeStorageType>* srcN;
00222         Markov::Node<NodeStorageType>* targetN;
00223         Markov::Edge<NodeStorageType>* e;
00224         if (this->nodes.find(src) == this->nodes.end()) {
00225             srcN = new Markov::Node<NodeStorageType>(src);
00226             this->nodes.insert(std::pair<char, Markov::Node<NodeStorageType>*>(src, srcN));
00227             //std::cout « "Creating new node at start.\n";
00228         }
00229         else {
00230             srcN = this->nodes.find(src)->second;
00231         }
00232
00233         if (this->nodes.find(target) == this->nodes.end()) {
00234             targetN = new Markov::Node<NodeStorageType>(target);
00235             this->nodes.insert(std::pair<char, Markov::Node<NodeStorageType>*>(target, targetN));
00236             //std::cout « "Creating new node at end.\n";
00237         }
00238         else {
00239             targetN = this->nodes.find(target)->second;
00240         }
00241         e = srcN->Link(targetN);
00242         e->AdjustEdge(oc);
00243         this->edges.push_back(e);
00244
00245         //std::cout « int(srcN->NodeValue()) « " --" « e->EdgeWeight() « "--> " «
      int(targetN->NodeValue()) « "\n";
00246
00247
00248     }
00249
00250     for (std::pair<unsigned char, Markov::Node<NodeStorageType>*> const& x : this->nodes) {
00251         //std::cout « "Total edges in EdgesV: " « x.second->edgesV.size() « "\n";
00252         std::sort (x.second->edgesV.begin(), x.second->edgesV.end(), [](Edge<NodeStorageType> *lhs,
      Edge<NodeStorageType> *rhs)->bool{
00253             return lhs->EdgeWeight() > rhs->EdgeWeight();
00254         });
00255         //for(int i=0;i<x.second->edgesV.size();i++)
00256         //  std::cout « x.second->edgesV[i]->EdgeWeight() « ", ";
00257         //std::cout « "\n";
00258     }
00259     //std::cout « "Total number of nodes: " « this->nodes.size() « std::endl;
00260     //std::cout « "Total number of edges: " « this->edges.size() « std::endl;
```

```
00261
00262     return true;
00263 }
```

8.6.2.21 ListCudaDevices()

```
static __host__ void Markov::API::CUDA::CUDADeviceController::ListCudaDevices ( )  [static],
[inherited]
```

List CUDA devices in the system.
This function will print details of every CUDA capable device in the system.
Example output:

```
Device Number: 0
Device name: GeForce RTX 2070
Memory Clock Rate (KHz): 7001000
Memory Bus Width (bits): 256
Peak Memory Bandwidth (GB/s): 448.064
Max Linear Threads: 1024
```

8.6.2.22 MigrateMatrix()

```
__host__ void Markov::API::CUDA::CUDAModelMatrix::MigrateMatrix ( )
```

Migrate the class members to the VRAM.
Cannot be used without calling Markov::API::ModelMatrix::ConstructMatrix at least once. This function will manage the memory allocation and data transfer from CPU RAM to GPU VRAM.
Newly allocated VRAM pointers are set in the class member variables.

8.6.2.23 Nodes()

```
std::map<char , Node<char >*>* Markov::Model< char >::Nodes ( )  [inline], [inherited]
```

Return starter Node.

Returns

starter node with 00 NodeValue

Definition at line 177 of file model.h.

```
00177 { return &nodes;}
```

8.6.2.24 OpenDatasetFile()

```
std::ifstream * Markov::API::MarkovPasswords::OpenDatasetFile (
            const char * filename )  [inherited]
```

Open dataset file and return the ifstream pointer.

Parameters

filename	- Filename to open

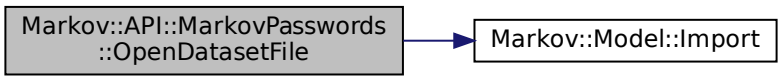
Returns

ifstream* to the the dataset file

Definition at line 43 of file markovPasswords.cpp.

```
00043                                                                    {
00044
00045     std::ifstream* datasetFile;
00046
00047     std::ifstream newFile(filename);
00048
00049     datasetFile = &newFile;
00050
00051     this->Import(datasetFile);
00052     return datasetFile;
00053 }
```

References Markov::Model< NodeStorageType >::Import().
Here is the call graph for this function:



8.6.2.25 RandomWalk()

```
char * Markov::Model< char >::RandomWalk (
            Markov::Random::RandomEngine * randomEngine,
            int minSetting,
            int maxSetting,
            NodeStorageType * buffer )  [inherited]
```

Do a random walk on this model.
Start from the starter node, on each node, invoke RandomNext using the random engine on current node, until terminator node is reached. If terminator node is reached before minimum length criateria is reached, ignore the last selection and re-invoke randomNext
If maximum length criteria is reached but final node is not, cut off the generation and proceed to the final node. This function takes Markov::Random::RandomEngine as a parameter to generate pseudo random numbers from
This library is shipped with two random engines, Marsaglia and Mersenne. While mersenne output is higher in entropy, most use cases don't really need super high entropy output, so Markov::Random::Marsaglia is preferable for better performance.
This function WILL NOT reallocate buffer. Make sure no out of bound writes are happening via maximum length criteria.
Example Use: Generate 10 lines, with 5 to 10 characters, and print the output. Use Marsaglia

```
Markov::Model<char> model;
Model.import("model.mdl");
char* res = new char[11];
Markov::Random::Marsaglia MarsagliaRandomEngine;
for (int i = 0; i < 10; i++) {
    this->RandomWalk(&MarsagliaRandomEngine, 5, 10, res);
    std::cout « res « "\n";
 }
```

Parameters

randomEngine	Random Engine to use for the random walks. For examples, see Markov::Random::Mersenne and Markov::Random::Marsaglia
minSetting	Minimum number of characters to generate
maxSetting	Maximum number of character to generate
buffer	buffer to write the result to

Returns

Null terminated string that was generated.

Definition at line 293 of file model.h.

```
00293                                                   {
00294     Markov::Node<NodeStorageType>* n = this->starterNode;
00295     int len = 0;
00296     Markov::Node<NodeStorageType>* temp_node;
00297     while (true) {
00298         temp_node = n->RandomNext(randomEngine);
00299         if (len >= maxSetting) {
00300             break;
```

```
00301         }
00302         else if ((temp_node == NULL) && (len < minSetting)) {
00303             continue;
00304         }
00305
00306         else if (temp_node == NULL){
00307             break;
00308         }
00309
00310         n = temp_node;
00311
00312         buffer[len++] = n->NodeValue();
00313     }
00314
00315     //null terminate the string
00316     buffer[len] = 0x00;
00317
00318     //do something with the generated string
00319     return buffer; //for now
00320 }
```

8.6.2.26 Save()

```
std::ofstream * Markov::API::MarkovPasswords::Save (
            const char * filename )  [inherited]
```

Export model to file.

Parameters

filename	- Export filename.

Returns

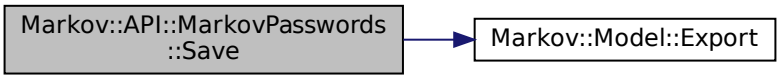
std::ofstream* of the exported file.

Definition at line 98 of file markovPasswords.cpp.

```
00098                                                                    {
00099     std::ofstream* exportFile;
00100
00101     std::ofstream newFile(filename);
00102
00103     exportFile = &newFile;
00104
00105     this->Export(exportFile);
00106     return exportFile;
00107 }
```

References Markov::Model< NodeStorageType >::Export().

Here is the call graph for this function:



8.6.2.27 StarterNode()

```
Node<char >* Markov::Model< char >::StarterNode ( )  [inline], [inherited]
```

Return starter Node.

Returns

starter node with 00 NodeValue

Definition at line 167 of file model.h.

```
00167 { return starterNode;}
```

8.6.2.28 Train()

```
void Markov::API::MarkovPasswords::Train (
            const char * datasetFileName,
            char delimiter,
            int threads )  [inherited]
```

Train the model with the dataset file.

Parameters

datasetFileName	- Ifstream* to the dataset. If null, use class member
delimiter	- a character, same as the delimiter in dataset content
threads	- number of OS threads to spawn

```
Markov::API::MarkovPasswords mp;
mp.Import("models/2gram.mdl");
mp.Train("password.corpus");
```

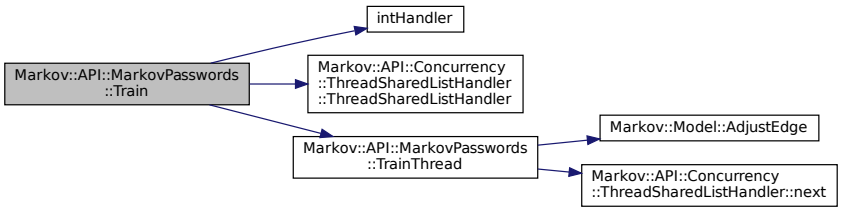
Definition at line 57 of file markovPasswords.cpp.

```
00057                                                                                                    {
00058   signal(SIGINT, intHandler);
00059     Markov::API::Concurrency::ThreadSharedListHandler listhandler(datasetFileName);
00060     auto start = std::chrono::high_resolution_clock::now();
00061
00062     std::vector<std::thread*> threadsV;
00063     for(int i=0;i<threads;i++){
00064         threadsV.push_back(new std::thread(&Markov::API::MarkovPasswords::TrainThread, this,
       &listhandler, delimiter));
00065     }
00066
00067     for(int i=0;i<threads;i++){
00068         threadsV[i]->join();
00069         delete threadsV[i];
00070     }
00071     auto finish = std::chrono::high_resolution_clock::now();
00072     std::chrono::duration<double> elapsed = finish - start;
00073     std::cout << "Elapsed time: " << elapsed.count() << " s\n";
00074
00075 }
```

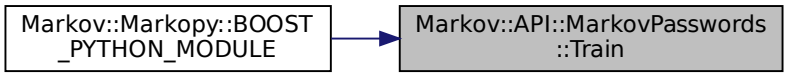
References intHandler(), Markov::API::Concurrency::ThreadSharedListHandler::ThreadSharedListHandler(), and Markov::API::MarkovPasswords::TrainThread().

Referenced by Markov::Markopy::BOOST_PYTHON_MODULE().

Here is the call graph for this function:



Here is the caller graph for this function:



8.6.2.29 TrainThread()

```
void Markov::API::MarkovPasswords::TrainThread (
            Markov::API::Concurrency::ThreadSharedListHandler * listhandler,
            char delimiter )  [private], [inherited]
```

A single thread invoked by the Train function.

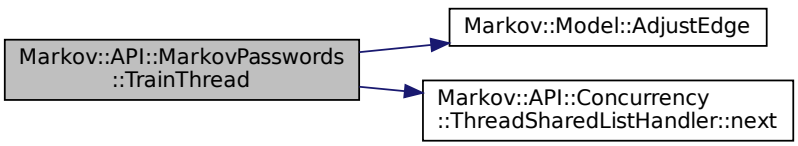
Parameters

listhandler	- Listhandler class to read corpus from
delimiter	- a character, same as the delimiter in dataset content

Definition at line 77 of file markovPasswords.cpp.

```
00077
                {
00078     char format_str[] ="%ld,%s";
00079     format_str[2]=delimiter;
00080     std::string line;
00081     while (listhandler->next(&line) && keepRunning) {
00082         long int oc;
00083         if (line.size() > 100) {
00084             line = line.substr(0, 100);
00085         }
00086         char* linebuf = new char[line.length()+5];
00087 #ifdef _WIN32
00088         sscanf_s(line.c_str(), "%ld,%s", &oc, linebuf, line.length()+5); //<== changed format_str to->
      "%ld,%s"
00089 #else
00090         sscanf(line.c_str(), format_str, &oc, linebuf);
00091 #endif
00092         this->AdjustEdge((const char*)linebuf, oc);
00093         delete linebuf;
00094     }
00095 }
```

References Markov::Model< NodeStorageType >::AdjustEdge(), keepRunning, and Markov::API::Concurrency::ThreadSharedListHa
Referenced by Markov::API::MarkovPasswords::Train().
Here is the call graph for this function:



Here is the caller graph for this function:

8.6.3 Member Data Documentation

8.6.3.1 datasetFile

`std::ifstream* Markov::API::MarkovPasswords::datasetFile [private], [inherited]`
Definition at line 106 of file markovPasswords.h.

8.6.3.2 device_edgeMatrix

`char* Markov::API::CUDA::CUDAModelMatrix::device_edgeMatrix [private]`
VRAM Address pointer of edge matrix (from modelMatrix.h)
Definition at line 73 of file cudaModelMatrix.h.

8.6.3.3 device_matrixIndex

`char* Markov::API::CUDA::CUDAModelMatrix::device_matrixIndex [private]`
VRAM Address pointer of matrixIndex (from modelMatrix.h)
Definition at line 83 of file cudaModelMatrix.h.

8.6.3.4 device_outputBuffer

`char* Markov::API::CUDA::CUDAModelMatrix::device_outputBuffer [private]`
RandomWalk results in device.
Definition at line 94 of file cudaModelMatrix.h.

8.6.3.5 device_totalEdgeWeights

`long int* Markov::API::CUDA::CUDAModelMatrix::device_totalEdgeWeights [private]`
VRAM Address pointer of total edge weights (from modelMatrix.h)
Definition at line 88 of file cudaModelMatrix.h.

8.6.3.6 device_valueMatrix

`long int* Markov::API::CUDA::CUDAModelMatrix::device_valueMatrix [private]`
VRAM Address pointer of value matrix (from modelMatrix.h)
Definition at line 78 of file cudaModelMatrix.h.

8.6.3.7 edgeMatrix

`char** Markov::API::ModelMatrix::edgeMatrix [protected], [inherited]`
2-D Character array for the edge Matrix (The characters of Nodes)
Definition at line 116 of file modelMatrix.h.

Referenced by Markov::API::ModelMatrix::ConstructMatrix(), Markov::API::ModelMatrix::DumpJSON(), and Markov::API::ModelMatrix::FastRandomWalkThread().

8.6.3.8 edges

`std::vector<Edge<char >*> Markov::Model< char >::edges [private], [inherited]`
A list of all edges in this model.
Definition at line 195 of file model.h.

8.6.3.9 flatEdgeMatrix

`char* Markov::API::CUDA::CUDAModelMatrix::flatEdgeMatrix [private]`
Adding Edge matrix end-to-end and resize to 1-D array for better perfomance on traversing.
Definition at line 104 of file cudaModelMatrix.h.

8.6.3.10 flatValueMatrix

`long int* Markov::API::CUDA::CUDAModelMatrix::flatValueMatrix [private]`
Adding Value matrix end-to-end and resize to 1-D array for better perfomance on traversing.
Definition at line 109 of file cudaModelMatrix.h.

8.6.3.11 matrixIndex

`char* Markov::API::ModelMatrix::matrixIndex [protected], [inherited]`
to hold the Matrix index (To hold the orders of 2-D arrays')
Definition at line 131 of file modelMatrix.h.
Referenced by Markov::API::ModelMatrix::ConstructMatrix(), Markov::API::ModelMatrix::DumpJSON(), and Markov::API::ModelMatrix::FastRandomWalkThread().

8.6.3.12 matrixSize

`int Markov::API::ModelMatrix::matrixSize [protected], [inherited]`
to hold Matrix size
Definition at line 126 of file modelMatrix.h.
Referenced by Markov::API::ModelMatrix::ConstructMatrix(), Markov::API::ModelMatrix::DumpJSON(), and Markov::API::ModelMatrix::FastRandomWalkThread().

8.6.3.13 modelSavefile

`std::ofstream* Markov::API::MarkovPasswords::modelSavefile [private], [inherited]`
Dataset file input of our system

Definition at line 107 of file markovPasswords.h.

8.6.3.14 nodes

`std::map<char , Node<char >*> Markov::Model< char >::nodes [private], [inherited]`
Map LeftNode is the Nodes NodeValue Map RightNode is the node pointer.
Definition at line 184 of file model.h.

8.6.3.15 outputBuffer

```
char* Markov::API::CUDA::CUDAModelMatrix::outputBuffer  [private]
```
RandomWalk results in host.
Definition at line 99 of file cudaModelMatrix.h.

8.6.3.16 outputFile

```
std::ofstream* Markov::API::MarkovPasswords::outputFile  [private], [inherited]
```
File to save model of our system

Definition at line 108 of file markovPasswords.h.

8.6.3.17 starterNode

```
Node<char >* Markov::Model< char >::starterNode  [private], [inherited]
```
Starter Node of this model.
Definition at line 189 of file model.h.

8.6.3.18 totalEdgeWeights

```
long int* Markov::API::ModelMatrix::totalEdgeWeights  [protected], [inherited]
```
Array of the Total Edge Weights.
Definition at line 136 of file modelMatrix.h.
Referenced by Markov::API::ModelMatrix::ConstructMatrix(), and Markov::API::ModelMatrix::FastRandomWalkThread().

8.6.3.19 valueMatrix

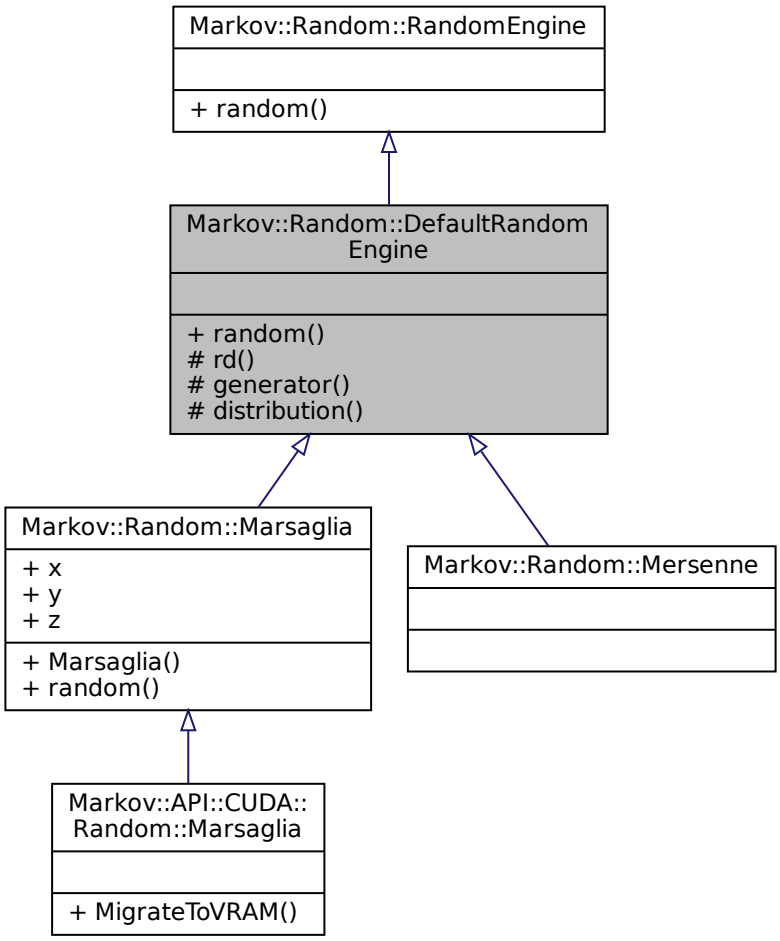
```
long int** Markov::API::ModelMatrix::valueMatrix  [protected], [inherited]
```
2-d Integer array for the value Matrix (For the weights of Edges)
Definition at line 121 of file modelMatrix.h.
Referenced by Markov::API::ModelMatrix::ConstructMatrix(), Markov::API::ModelMatrix::DumpJSON(), and Markov::API::ModelMatrix::FastRandomWalkThread().
The documentation for this class was generated from the following file:

- cudaModelMatrix.h

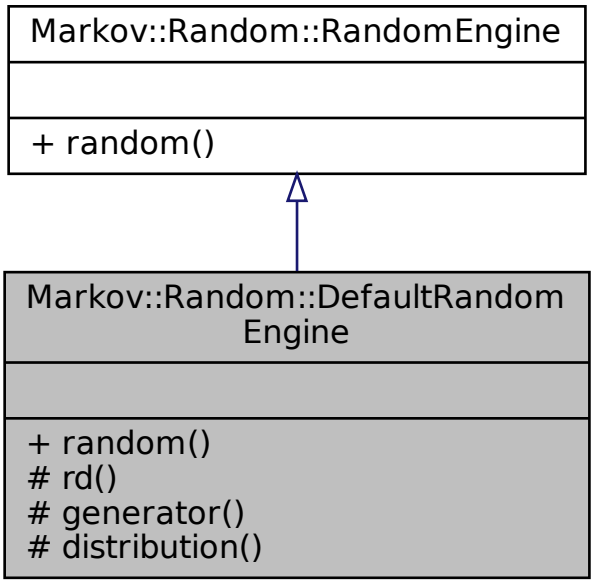
8.7 Markov::Random::DefaultRandomEngine Class Reference

Implementation using Random.h default random engine.
`#include <random.h>`

Inheritance diagram for Markov::Random::DefaultRandomEngine:



Collaboration diagram for Markov::Random::DefaultRandomEngine:



Public Member Functions

- unsigned long random ()

  Generate Random Number.

Protected Member Functions

- std::random_device & rd ()

  Default random device for seeding.
- std::default_random_engine & generator ()

  Default random engine for seeding.
- std::uniform_int_distribution< long long unsigned > & distribution ()

  Distribution schema for seeding.

8.7.1 Detailed Description

Implementation using Random.h default random engine.
This engine is also used by other engines for seeding.
Example Use: Using Default Engine with RandomWalk

```
Markov::Model<char> model;
Model.import("model.mdl");
char* res = new char[11];
Markov::Random::DefaultRandomEngine randomEngine;
for (int i = 0; i < 10; i++) {
    this->RandomWalk(&randomEngine, 5, 10, res);
    std::cout « res « "\n";
 }
```

Example Use: Generating a random number with Marsaglia Engine

```
Markov::Random::DefaultRandomEngine de;
std::cout « de.random();
```

Definition at line 52 of file random.h.

8.7.2 Member Function Documentation

8.7.2.1 distribution()

```
std::uniform_int_distribution<long long unsigned>& Markov::Random::DefaultRandomEngine::distribution
( )  [inline], [protected]
```

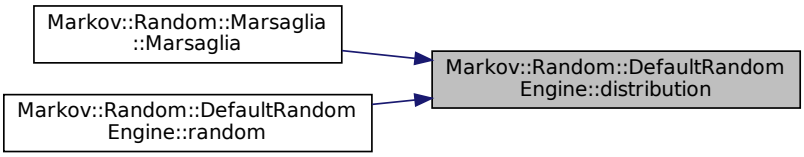
Distribution schema for seeding.

Definition at line 81 of file random.h.

```
00081                                                                                {
00082             static std::uniform_int_distribution<long long unsigned> _distribution(0, 0xffffFFFF);
00083             return _distribution;
00084         }
```

Referenced by Markov::Random::Marsaglia::Marsaglia(), and random().

Here is the caller graph for this function:



8.7.2.2 generator()

```
std::default_random_engine& Markov::Random::DefaultRandomEngine::generator ( )  [inline],
[protected]
```

Default random engine for seeding.

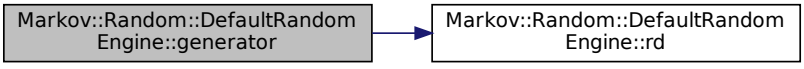
Definition at line 73 of file random.h.

```
00073                                                {
00074             static std::default_random_engine _generator(rd()());
00075             return _generator;
00076         }
```

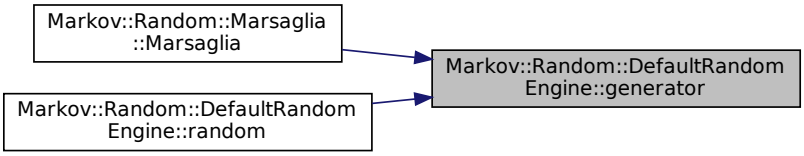
References rd().

Referenced by Markov::Random::Marsaglia::Marsaglia(), and random().

Here is the call graph for this function:



Here is the caller graph for this function:



8.7.2.3 random()

```
unsigned long Markov::Random::DefaultRandomEngine::random ( )  [inline], [virtual]
```

Generate Random Number.

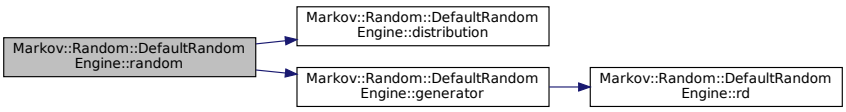
Returns

random number in long range.

Implements Markov::Random::RandomEngine.
Reimplemented in Markov::Random::Marsaglia.
Definition at line 57 of file random.h.

```
00057                                   {
00058             return this->distribution()(this->generator());
00059         }
```

References distribution(), and generator().
Here is the call graph for this function:



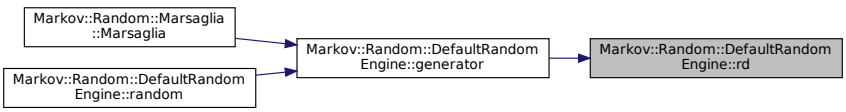
8.7.2.4 rd()

```
std::random_device& Markov::Random::DefaultRandomEngine::rd ( )  [inline], [protected]
```

Default random device for seeding.
Definition at line 65 of file random.h.

```
00065                                    {
00066             static std::random_device _rd;
00067             return _rd;
00068         }
```

Referenced by generator().
Here is the caller graph for this function:

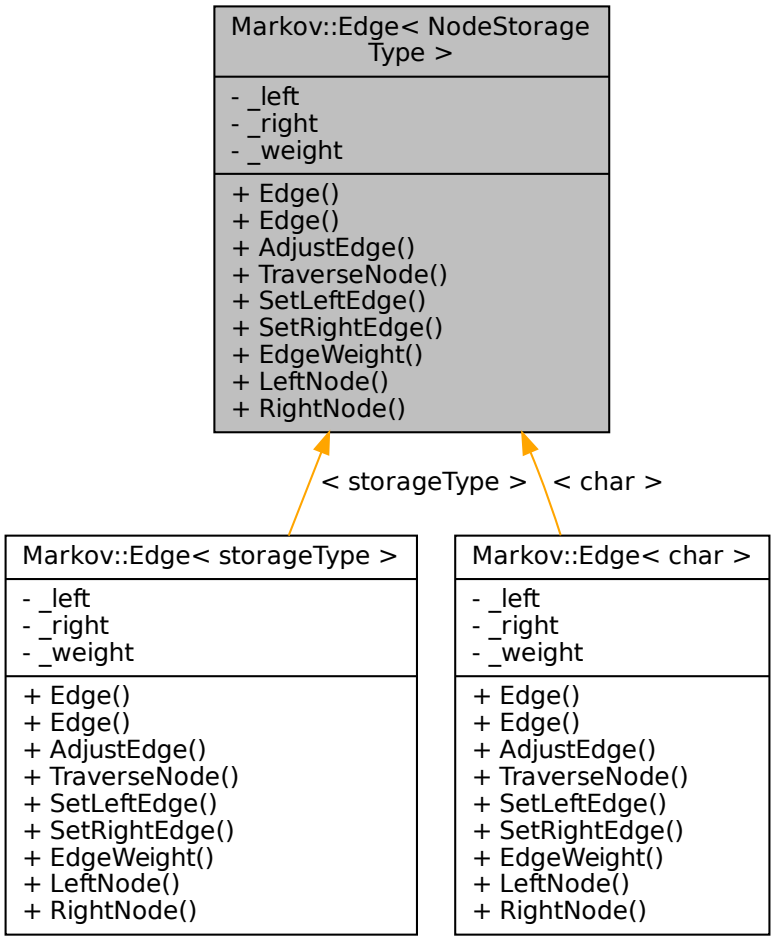


The documentation for this class was generated from the following file:

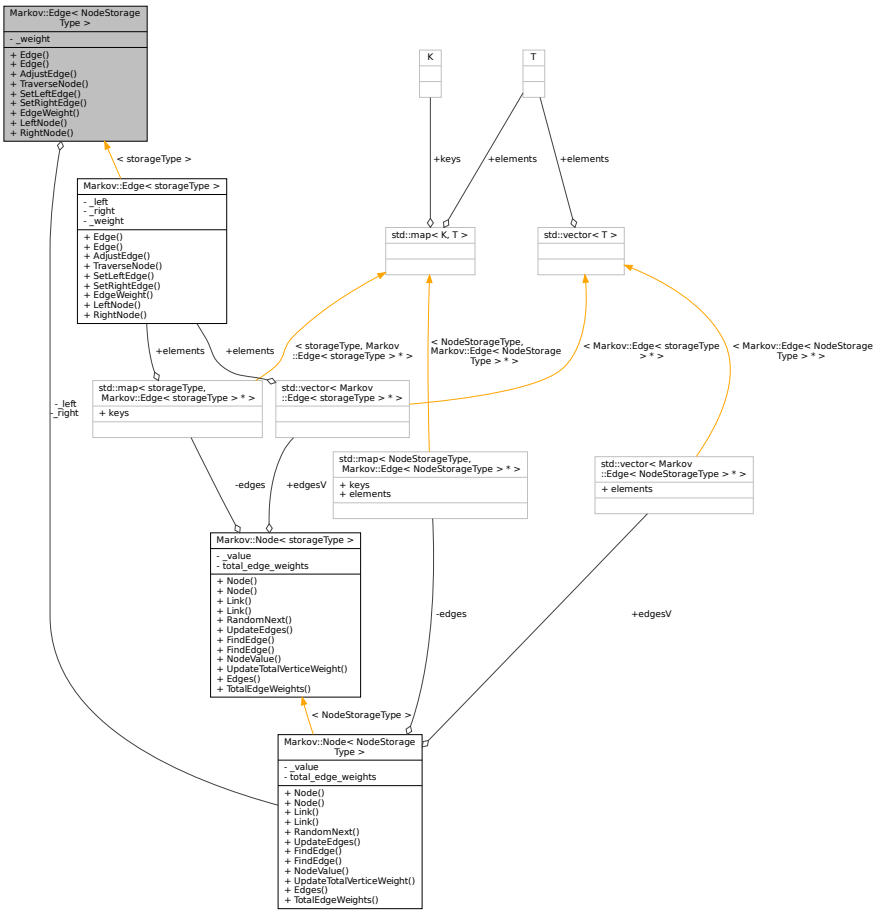
- random.h

8.8 Markov::Edge< NodeStorageType > Class Template Reference

Edge class used to link nodes in the model together.
`#include <model.h>`
Inheritance diagram for Markov::Edge< NodeStorageType >:



Collaboration diagram for Markov::Edge< NodeStorageType >:



Public Member Functions

- Edge ()

  Default constructor.
- Edge (Node< NodeStorageType > *_left, Node< NodeStorageType > *_right)

  Constructor. Initialize edge with given RightNode and LeftNode.
- void AdjustEdge (long int offset)

  Adjust the edge EdgeWeight with offset. Adds the offset parameter to the edge EdgeWeight.
- Node< NodeStorageType > * TraverseNode ()

  Traverse this edge to RightNode.
- void SetLeftEdge (Node< NodeStorageType > *)

  Set LeftNode of this edge.
- void SetRightEdge (Node< NodeStorageType > *)

  Set RightNode of this edge.
- uint64_t EdgeWeight ()

  return edge's EdgeWeight.
- Node< NodeStorageType > * LeftNode ()

  return edge's LeftNode
- Node< NodeStorageType > * RightNode ()

  return edge's RightNode

Private Attributes

- Node< NodeStorageType > * _left

  source node
- Node< NodeStorageType > * _right

  target node
- long int _weight

  Edge Edge Weight.

8.8.1 Detailed Description

template<typename NodeStorageType>
class Markov::Edge< NodeStorageType >

Edge class used to link nodes in the model together.
Has LeftNode, RightNode, and EdgeWeight of the edge. Edges are *UNIDIRECTIONAL* in this model. They can only be traversed LeftNode to RightNode.
Definition at line 26 of file model.h.

8.8.2 Constructor & Destructor Documentation

8.8.2.1 Edge() [1/2]

```
template<typename NodeStorageType >
Markov::Edge< NodeStorageType >::Edge
```

Default constructor.
Definition at line 116 of file edge.h.

```
00116                                   {
00117     this->_left = NULL;
00118     this->_right = NULL;
00119     this->_weight = 0;
00120 }
```

8.8.2.2 Edge() [2/2]

```
template<typename NodeStorageType >
Markov::Edge< NodeStorageType >::Edge (
            Markov::Node< NodeStorageType > * _left,
            Markov::Node< NodeStorageType > * _right )
```

Constructor. Initialize edge with given RightNode and LeftNode.

Parameters

_left	- Left node of this edge.
_right	- Right node of this edge.

Example Use: Construct edge

```
Markov::Node<unsigned char>* src = new Markov::Node<unsigned char>('a');
Markov::Node<unsigned char>* target1 = new Markov::Node<unsigned char>('b');
Markov::Edge<unsigned char>* e1 = new Markov::Edge<unsigned char>(src, target1);
```

Definition at line 123 of file edge.h.

```
00123
        {
00124     this->_left = _left;
00125     this->_right = _right;
00126     this->_weight = 0;
00127 }
```

8.8.3 Member Function Documentation

8.8.3.1 AdjustEdge()

```
template<typename NodeStorageType >
void Markov::Edge< NodeStorageType >::AdjustEdge (
            long int offset )
```

Adjust the edge EdgeWeight with offset. Adds the offset parameter to the edge EdgeWeight.

Parameters

offset	- NodeValue to be added to the EdgeWeight

Example Use: Construct edge

```
Markov::Node<unsigned char>* src = new Markov::Node<unsigned char>('a');
Markov::Node<unsigned char>* target1 = new Markov::Node<unsigned char>('b');
Markov::Edge<unsigned char>* e1 = new Markov::Edge<unsigned char>(src, target1);
e1->AdjustEdge(25);
```

Definition at line 130 of file edge.h.

```
00130                                                          {
00131     this->_weight += offset;
00132     this->LeftNode()->UpdateTotalVerticeWeight(offset);
00133 }
```

8.8.3.2 EdgeWeight()

```
template<typename NodeStorageType >
uint64_t Markov::Edge< NodeStorageType >::EdgeWeight  [inline]
```

return edge's EdgeWeight.

Returns

edge's EdgeWeight.

Definition at line 153 of file edge.h.

```
00153                                                  {
00154     return this->_weight;
00155 }
```

Referenced by Markov::API::ModelMatrix::ConstructMatrix().

Here is the caller graph for this function:

8.8.3.3 LeftNode()

```
template<typename NodeStorageType >
Markov::Node< NodeStorageType > * Markov::Edge< NodeStorageType >::LeftNode
```

return edge's LeftNode

Returns

edge's LeftNode.

Definition at line 158 of file edge.h.

```
00158                                                                    {
00159     return this->_left;
00160 }
```

8.8.3.4 RightNode()

```
template<typename NodeStorageType >
Markov::Node< NodeStorageType > * Markov::Edge< NodeStorageType >::RightNode  [inline]
```

return edge's RightNode

Returns

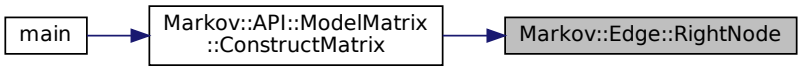
edge's RightNode.

Definition at line 163 of file edge.h.

```
00163                                                                     {
00164     return this->_right;
00165 }
```

Referenced by Markov::API::ModelMatrix::ConstructMatrix().

Here is the caller graph for this function:



8.8.3.5 SetLeftEdge()

```
template<typename NodeStorageType >
void Markov::Edge< NodeStorageType >::SetLeftEdge (
            Markov::Node< NodeStorageType > * n )
```

Set LeftNode of this edge.

Parameters

node	- Node to be linked with.

Definition at line 143 of file edge.h.

```
00143                                                                     {
00144     this->_left = n;
00145 }
```

8.8.3.6 SetRightEdge()

```
template<typename NodeStorageType >
void Markov::Edge< NodeStorageType >::SetRightEdge (
            Markov::Node< NodeStorageType > * n )
```

Set RightNode of this edge.

Parameters

node	- Node to be linked with.

Definition at line 148 of file edge.h.

```
00148                                                                    {
00149     this->_right = n;
00150 }
```

8.8.3.7 TraverseNode()

```
template<typename NodeStorageType >
Markov::Node< NodeStorageType > * Markov::Edge< NodeStorageType >::TraverseNode  [inline]
```

Traverse this edge to RightNode.

Returns

Right node. If this is a terminator node, return NULL

Example Use: Traverse a node

```
Markov::Node<unsigned char>* src = new Markov::Node<unsigned char>('a');
Markov::Node<unsigned char>* target1 = new Markov::Node<unsigned char>('b');
Markov::Edge<unsigned char>* e1 = new Markov::Edge<unsigned char>(src, target1);
e1->AdjustEdge(25);
Markov::Edge<unsigned char>* e2 = e1->traverseNode();
```

Definition at line 136 of file edge.h.

```
00136                                                                    {
00137     if (this->RightNode()->NodeValue() == 0xff) //terminator node
00138         return NULL;
00139     return _right;
00140 }
```

8.8.4 Member Data Documentation

8.8.4.1 _left

```
template<typename NodeStorageType >
Node<NodeStorageType>* Markov::Edge< NodeStorageType >::_left  [private]
```

source node

Definition at line 98 of file edge.h.

8.8.4.2 _right

```
template<typename NodeStorageType >
Node<NodeStorageType>* Markov::Edge< NodeStorageType >::_right  [private]
```

target node

Definition at line 103 of file edge.h.

Referenced by Markov::Edge< char >::TraverseNode().

8.8.4.3 _weight

```
template<typename NodeStorageType >
long int Markov::Edge< NodeStorageType >::_weight  [private]
```

Edge Edge Weight.

Definition at line 108 of file edge.h.

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

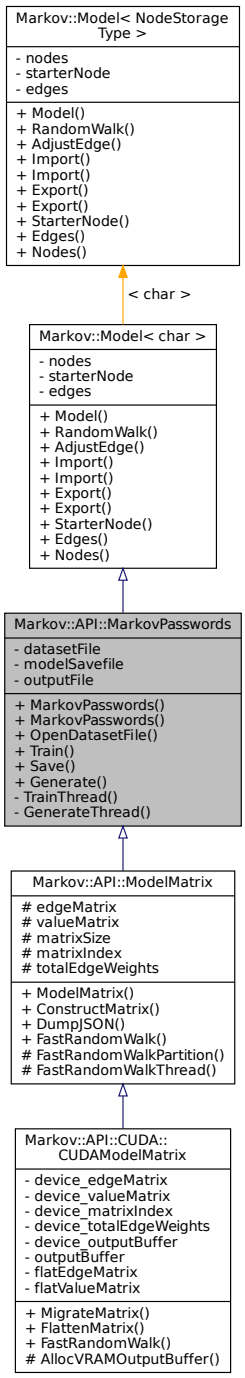
- model.h
- edge.h

8.9 Markov::API::MarkovPasswords Class Reference

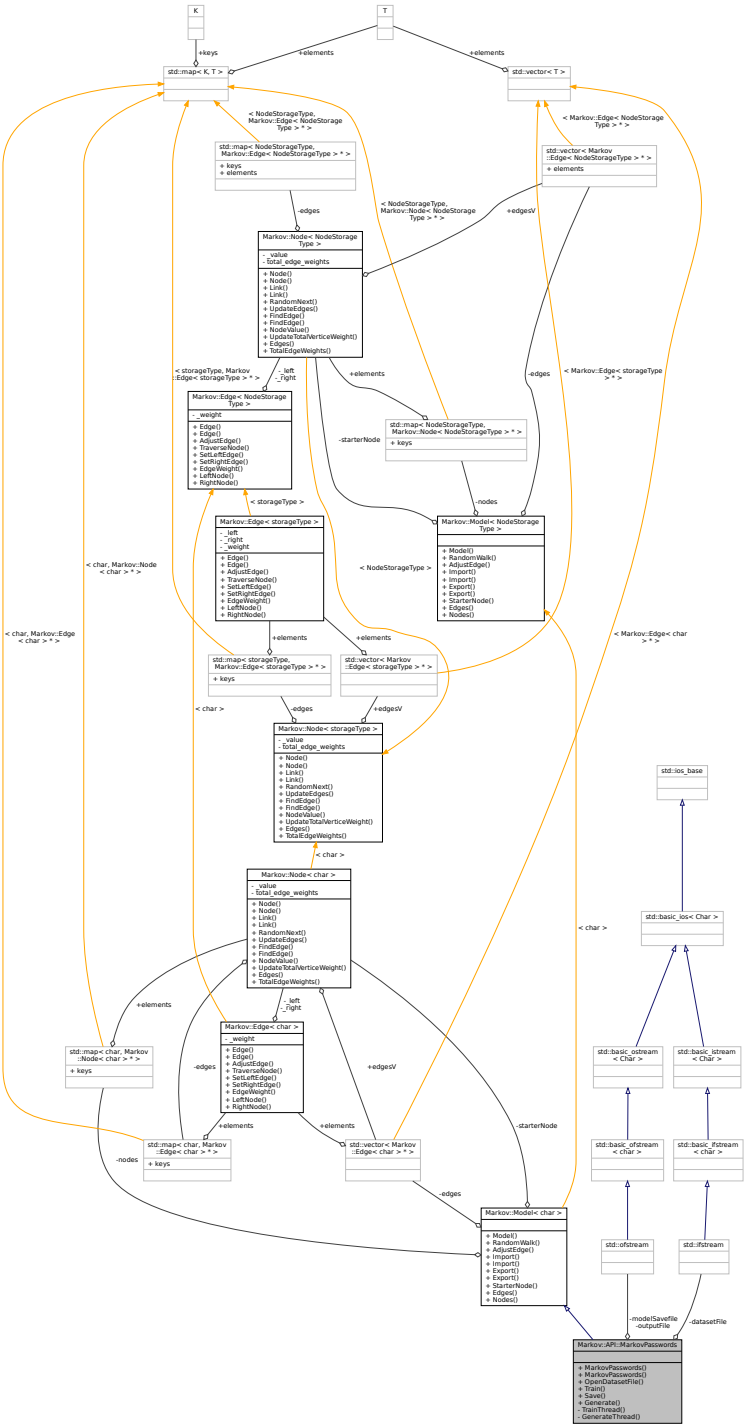
Markov::Model with char represented nodes.

`#include <markovPasswords.h>`

Inheritance diagram for Markov::API::MarkovPasswords:



Collaboration diagram for Markov::API::MarkovPasswords:



Public Member Functions

- MarkovPasswords ()

  Initialize the markov model from MarkovModel::Markov::Model.
- MarkovPasswords (const char ∗filename)

  Initialize the markov model from MarkovModel::Markov::Model, with an import file.
- std::ifstream ∗ OpenDatasetFile (const char ∗filename)

  Open dataset file and return the ifstream pointer.

- void Train (const char *datasetFileName, char delimiter, int threads)

  Train the model with the dataset file.
- std::ofstream * Save (const char *filename)

  Export model to file.
- void Generate (unsigned long int n, const char *wordlistFileName, int minLen=6, int maxLen=12, int threads=20)

  Call Markov::Model::RandomWalk n times, and collect output.
- char * RandomWalk (Markov::Random::RandomEngine *randomEngine, int minSetting, int maxSetting, char *buffer)

  Do a random walk on this model.
- void AdjustEdge (const char *payload, long int occurrence)

  Adjust the model with a single string.
- bool Import (std::ifstream *)

  Import a file to construct the model.
- bool Import (const char *filename)

  Open a file to import with filename, and call bool Model::Import with std::ifstream.
- bool Export (std::ofstream *)

  Export a file of the model.
- bool Export (const char *filename)

  Open a file to export with filename, and call bool Model::Export with std::ofstream.
- Node< char > * StarterNode ()

  Return starter Node.
- std::vector< Edge< char > * > * Edges ()

  Return a vector of all the edges in the model.
- std::map< char, Node< char > * > * Nodes ()

  Return starter Node.

Private Member Functions

- void TrainThread (Markov::API::Concurrency::ThreadSharedListHandler *listhandler, char delimiter)

  A single thread invoked by the Train function.
- void GenerateThread (std::mutex *outputLock, unsigned long int n, std::ofstream *wordlist, int minLen, int maxLen)

  A single thread invoked by the Generate function.

Private Attributes

- std::ifstream * datasetFile
- std::ofstream * modelSavefile

  Dataset file input of our system
- std::ofstream * outputFile

  File to save model of our system
- std::map< char, Node< char > * > nodes

  Map LeftNode is the Nodes NodeValue Map RightNode is the node pointer.
- Node< char > * starterNode

  Starter Node of this model.
- std::vector< Edge< char > * > edges

  A list of all edges in this model.

8.9.1 Detailed Description

Markov::Model with char represented nodes.
Includes wrappers for Markov::Model and additional helper functions to handle file I/O
This class is an extension of Markov::Model<char>, with higher level abstractions such as train and generate.
Definition at line 17 of file markovPasswords.h.

8.9.2 Constructor & Destructor Documentation

8.9.2.1 MarkovPasswords() [1/2]

```
Markov::API::MarkovPasswords::MarkovPasswords ( )
```

Initialize the markov model from MarkovModel::Markov::Model.
Parent constructor. Has no extra functionality.
Definition at line 26 of file markovPasswords.cpp.

```
00026                                           : Markov::Model<char>(){
00027
00028
00029 }
```

8.9.2.2 MarkovPasswords() [2/2]

```
Markov::API::MarkovPasswords::MarkovPasswords (
            const char * filename )
```

Initialize the markov model from MarkovModel::Markov::Model, with an import file.
This function calls the Markov::Model::Import on the filename to construct the model. Same thing as creating and empty model, and calling MarkovPasswords::Import on the filename.

Parameters

filename	- Filename to import

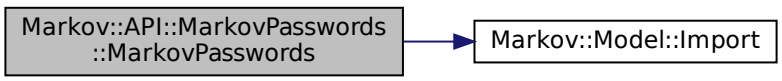
Example Use: Construction via filename

```
MarkovPasswords mp("test.mdl");
```

Definition at line 31 of file markovPasswords.cpp.

```
00031                                                                   {
00032
00033     std::ifstream* importFile;
00034
00035     this->Import(filename);
00036
00037     //std::ifstream* newFile(filename);
00038
00039     //importFile = newFile;
00040
00041 }
```

References Markov::Model< NodeStorageType >::Import().
Here is the call graph for this function:



8.9.3 Member Function Documentation

8.9.3.1 AdjustEdge()

```
void Markov::Model< char >::AdjustEdge (
            const char * payload,
            long int occurrence )  [inherited]
```

Adjust the model with a single string.
Start from the starter node, and for each character, AdjustEdge the edge EdgeWeight from current node to the next, until NULL character is reached.
Then, update the edge EdgeWeight from current node, to the terminator node.
This function is used for training purposes, as it can be used for adjusting the model with each line of the corpus file.
Example Use: Create an empty model and train it with string: "testdata"

```
Markov::Model<char> model;
char test[] = "testdata";
model.AdjustEdge(test, 15);
```

Parameters

string	- String that is passed from the training, and will be used to AdjustEdge the model with
occurrence	- Occurrence of this string.

Definition at line 323 of file model.h.

```
00323                                                                                                {
00324     NodeStorageType p = payload[0];
00325     Markov::Node<NodeStorageType>* curnode = this->starterNode;
00326     Markov::Edge<NodeStorageType>* e;
00327     int i = 0;
00328
00329     if (p == 0) return;
00330     while (p != 0) {
00331         e = curnode->FindEdge(p);
00332         if (e == NULL) return;
00333         e->AdjustEdge(occurrence);
00334         curnode = e->RightNode();
00335         p = payload[++i];
00336     }
00337
00338     e = curnode->FindEdge('\xff');
00339     e->AdjustEdge(occurrence);
00340     return;
00341 }
```

8.9.3.2 Edges()

```
std::vector<Edge<char >*>* Markov::Model< char >::Edges  [inline], [inherited]
```

Return a vector of all the edges in the model.

Returns

vector of edges

Definition at line 172 of file model.h.

```
00172 { return &edges;}
```

8.9.3.3 Export() [1/2]

```
bool Markov::Model< char >::Export (
            const char * filename )  [inherited]
```

Open a file to export with filename, and call bool Model::Export with std::ofstream.

Returns

True if successful, False for incomplete models or corrupt file formats

Example Use: Export file to filename

```
Markov::Model<char> model;
model.Export("test.mdl");
```

Definition at line 286 of file model.h.

```
00286                                                   {
00287     std::ofstream exportfile;
00288     exportfile.open(filename);
00289     return this->Export(&exportfile);
00290 }
```

8.9.3.4 Export() [2/2]

```
bool Markov::Model< char >::Export (
            std::ofstream * f )  [inherited]
```

Export a file of the model.
File contains a list of edges. Format is: Left_repr;EdgeWeight;right_repr. For more information on the format, check out the project wiki or github readme.
Iterate over this vertices, and their edges, and write them to file.

Returns

True if successful, False for incomplete models.

Example Use: Export file to ofstream

```
Markov::Model<char> model;
std::ofstream file("test.mdl");
model.Export(&file);
```

Definition at line 274 of file model.h.

```
00274                                                   {
00275     Markov::Edge<NodeStorageType>* e;
00276     for (std::vector<int>::size_type i = 0; i != this->edges.size(); i++) {
00277         e = this->edges[i];
00278         //std::cout « e->LeftNode()->NodeValue() « "," « e->EdgeWeight() « "," «
      e->RightNode()->NodeValue() « "\n";
00279         *f « e->LeftNode()->NodeValue() « "," « e->EdgeWeight() « "," « e->RightNode()->NodeValue() «
      "\n";
00280     }
00281
00282     return true;
00283 }
```

8.9.3.5 Generate()

```
void Markov::API::MarkovPasswords::Generate (
            unsigned long int n,
            const char * wordlistFileName,
            int minLen = 6,
            int maxLen = 12,
            int threads = 20 )
```

Call Markov::Model::RandomWalk n times, and collect output.
Generate from model and write results to a file. a much more performance-optimized method. FastRandomWalk will reduce the runtime by %96.5 on average.

Deprecated See Markov::API::MatrixModel::FastRandomWalk for more information.

Parameters

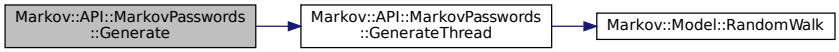
n	- Number of passwords to generate.
wordlistFileName	- Filename to write to
minLen	- Minimum password length to generate
maxLen	- Maximum password length to generate
threads	- number of OS threads to spawn

Definition at line 110 of file markovPasswords.cpp.

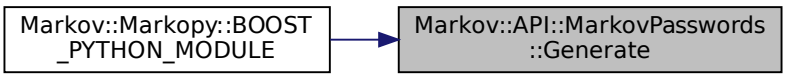
```
00110                                                   {
00111     char* res;
```

```
00112     char print[100];
00113     std::ofstream wordlist;
00114     wordlist.open(wordlistFileName);
00115     std::mutex mlock;
00116     int iterationsPerThread = n/threads;
00117     int iterationsCarryOver = n%threads;
00118     std::vector<std::thread*> threadsV;
00119     for(int i=0;i<threads;i++){
00120         threadsV.push_back(new std::thread(&Markov::API::MarkovPasswords::GenerateThread, this,
      &mlock, iterationsPerThread, &wordlist, minLen, maxLen));
00121     }
00122
00123     for(int i=0;i<threads;i++){
00124         threadsV[i]->join();
00125         delete threadsV[i];
00126     }
00127
00128     this->GenerateThread(&mlock, iterationsCarryOver, &wordlist, minLen, maxLen);
00129
00130 }
```

References GenerateThread().
Referenced by Markov::Markopy::BOOST_PYTHON_MODULE().
Here is the call graph for this function:



Here is the caller graph for this function:



8.9.3.6 GenerateThread()

```
void Markov::API::MarkovPasswords::GenerateThread (
            std::mutex * outputLock,
            unsigned long int n,
            std::ofstream * wordlist,
            int minLen,
            int maxLen )  [private]
```

A single thread invoked by the Generate function.
DEPRECATED: See Markov::API::MatrixModel::FastRandomWalkThread for more information. This has been replaced with a much more performance-optimized method. FastRandomWalk will reduce the runtime by %96.5 on average.

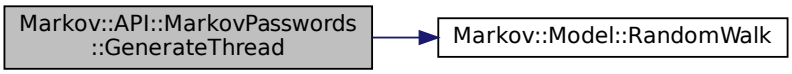
Parameters

outputLock	- shared mutex lock to lock during output operation. Prevents race condition on write.
n	number of lines to be generated by this thread
wordlist	wordlistfile
minLen	- Minimum password length to generate
maxLen	- Maximum password length to generate

Definition at line 132 of file markovPasswords.cpp.

```
00132                                       {
00133     char* res = new char[maxLen+5];
00134     if(n==0) return;
00135
00136     Markov::Random::Marsaglia MarsagliaRandomEngine;
00137     for (int i = 0; i < n; i++) {
00138         this->RandomWalk(&MarsagliaRandomEngine, minLen, maxLen, res);
00139         outputLock->lock();
00140         *wordlist << res << "\n";
00141         outputLock->unlock();
00142     }
00143 }
```

References Markov::Model< NodeStorageType >::RandomWalk().
Referenced by Generate().
Here is the call graph for this function:



Here is the caller graph for this function:

8.9.3.7 Import() [1/2]

```
bool Markov::Model< char >::Import (
            const char * filename )  [inherited]
```

Open a file to import with filename, and call bool Model::Import with std::ifstream.

Returns

True if successful, False for incomplete models or corrupt file formats

Example Use: Import a file with filename

```
Markov::Model<char> model;
model.Import("test.mdl");
```

Definition at line 266 of file model.h.

```
00266                                                       {
00267     std::ifstream importfile;
00268     importfile.open(filename);
00269     return this->Import(&importfile);
00270
00271 }
```

8.9.3.8 Import() [2/2]

```
bool Markov::Model< char >::Import (
            std::ifstream * f )  [inherited]
```

Import a file to construct the model.
File contains a list of edges. For more info on the file format, check out the wiki and github readme pages. Format is: Left_repr;EdgeWeight;right_repr
Iterate over this list, and construct nodes and edges accordingly.

Returns

True if successful, False for incomplete models or corrupt file formats

Example Use: Import a file from ifstream

```
Markov::Model<char> model;
std::ifstream file("test.mdl");
model.Import(&file);
```

Definition at line 207 of file model.h.

```
00207                                                   {
00208     std::string cell;
00209
00210     char src;
00211     char target;
00212     long int oc;
00213
00214     while (std::getline(*f, cell)) {
00215         //std::cout « "cell: " « cell « std::endl;
00216         src = cell[0];
00217         target = cell[cell.length() - 1];
00218         char* j;
00219         oc = std::strtol(cell.substr(2, cell.length() - 2).c_str(),&j,10);
00220         //std::cout « oc « "\n";
00221         Markov::Node<NodeStorageType>* srcN;
00222         Markov::Node<NodeStorageType>* targetN;
00223         Markov::Edge<NodeStorageType>* e;
00224         if (this->nodes.find(src) == this->nodes.end()) {
00225             srcN = new Markov::Node<NodeStorageType>(src);
00226             this->nodes.insert(std::pair<char, Markov::Node<NodeStorageType>*>(src, srcN));
00227             //std::cout « "Creating new node at start.\n";
00228         }
00229         else {
00230             srcN = this->nodes.find(src)->second;
00231         }
00232
00233         if (this->nodes.find(target) == this->nodes.end()) {
00234             targetN = new Markov::Node<NodeStorageType>(target);
00235             this->nodes.insert(std::pair<char, Markov::Node<NodeStorageType>*>(target, targetN));
00236             //std::cout « "Creating new node at end.\n";
00237         }
00238         else {
00239             targetN = this->nodes.find(target)->second;
00240         }
00241         e = srcN->Link(targetN);
00242         e->AdjustEdge(oc);
00243         this->edges.push_back(e);
00244
00245         //std::cout « int(srcN->NodeValue()) « " --" « e->EdgeWeight() « "--> " «
      int(targetN->NodeValue()) « "\n";
00246
00247
00248     }
00249
00250     for (std::pair<unsigned char, Markov::Node<NodeStorageType>*> const& x : this->nodes) {
00251         //std::cout « "Total edges in EdgesV: " « x.second->edgesV.size() « "\n";
00252         std::sort (x.second->edgesV.begin(), x.second->edgesV.end(), [](Edge<NodeStorageType> *lhs,
      Edge<NodeStorageType> *rhs)->bool{
00253             return lhs->EdgeWeight() > rhs->EdgeWeight();
00254         });
00255         //for(int i=0;i<x.second->edgesV.size();i++)
00256         //  std::cout « x.second->edgesV[i]->EdgeWeight() « ", ";
00257         //std::cout « "\n";
00258     }
00259     //std::cout « "Total number of nodes: " « this->nodes.size() « std::endl;
00260     //std::cout « "Total number of edges: " « this->edges.size() « std::endl;
00261
00262     return true;
00263 }
```

8.9.3.9 Nodes()

```
std::map<char , Node<char >*>* Markov::Model< char >::Nodes  [inline], [inherited]
```

Return starter Node.

Returns

starter node with 00 NodeValue

Definition at line 177 of file model.h.

```
00177 { return &nodes;}
```

8.9.3.10 OpenDatasetFile()

```
std::ifstream * Markov::API::MarkovPasswords::OpenDatasetFile (
            const char * filename )
```

Open dataset file and return the ifstream pointer.

Parameters

filename	- Filename to open

Returns

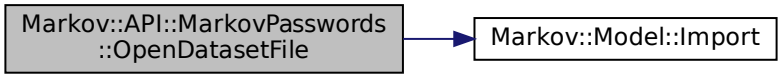
ifstream* to the the dataset file

Definition at line 43 of file markovPasswords.cpp.

```
00043                                                                    {
00044
00045     std::ifstream* datasetFile;
00046
00047     std::ifstream newFile(filename);
00048
00049     datasetFile = &newFile;
00050
00051     this->Import(datasetFile);
00052     return datasetFile;
00053 }
```

References Markov::Model< NodeStorageType >::Import().

Here is the call graph for this function:



8.9.3.11 RandomWalk()

```
char * Markov::Model< char >::RandomWalk (
            Markov::Random::RandomEngine * randomEngine,
            int minSetting,
            int maxSetting,
            char * buffer )  [inherited]
```

Do a random walk on this model.

Start from the starter node, on each node, invoke RandomNext using the random engine on current node, until terminator node is reached. If terminator node is reached before minimum length criateria is reached, ignore the last selection and re-invoke randomNext

If maximum length criteria is reached but final node is not, cut off the generation and proceed to the final node. This function takes Markov::Random::RandomEngine as a parameter to generate pseudo random numbers from

This library is shipped with two random engines, Marsaglia and Mersenne. While mersenne output is higher in entropy, most use cases don't really need super high entropy output, so Markov::Random::Marsaglia is preferable for better performance.

This function WILL NOT reallocate buffer. Make sure no out of bound writes are happening via maximum length criteria.

Example Use: Generate 10 lines, with 5 to 10 characters, and print the output. Use Marsaglia

```
Markov::Model<char> model;
Model.import("model.mdl");
char* res = new char[11];
Markov::Random::Marsaglia MarsagliaRandomEngine;
for (int i = 0; i < 10; i++) {
```

```
    this->RandomWalk(&MarsagliaRandomEngine, 5, 10, res);
    std::cout « res « "\n";
}
```

Parameters

randomEngine	Random Engine to use for the random walks. For examples, see Markov::Random::Mersenne and Markov::Random::Marsaglia
minSetting	Minimum number of characters to generate
maxSetting	Maximum number of character to generate
buffer	buffer to write the result to

Returns

Null terminated string that was generated.

Definition at line 293 of file model.h.

```
00293                                                                {
00294     Markov::Node<NodeStorageType>* n = this->starterNode;
00295     int len = 0;
00296     Markov::Node<NodeStorageType>* temp_node;
00297     while (true) {
00298         temp_node = n->RandomNext(randomEngine);
00299         if (len >= maxSetting) {
00300             break;
00301         }
00302         else if ((temp_node == NULL) && (len < minSetting)) {
00303             continue;
00304         }
00305
00306         else if (temp_node == NULL){
00307             break;
00308         }
00309
00310         n = temp_node;
00311
00312         buffer[len++] = n->NodeValue();
00313     }
00314
00315     //null terminate the string
00316     buffer[len] = 0x00;
00317
00318     //do something with the generated string
00319     return buffer; //for now
00320 }
```

8.9.3.12 Save()

```
std::ofstream * Markov::API::MarkovPasswords::Save (
            const char * filename )
```

Export model to file.

Parameters

filename	- Export filename.

Returns

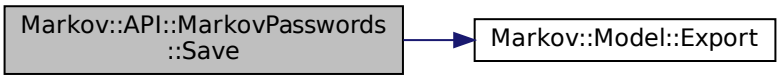
std::ofstream* of the exported file.

Definition at line 98 of file markovPasswords.cpp.

```
00098                                                                {
00099     std::ofstream* exportFile;
00100
00101     std::ofstream newFile(filename);
00102
00103     exportFile = &newFile;
00104
00105     this->Export(exportFile);
```

```
00106     return exportFile;
00107 }
```

References Markov::Model< NodeStorageType >::Export().
Here is the call graph for this function:



8.9.3.13 StarterNode()

```
Node<char >* Markov::Model< char >::StarterNode  [inline], [inherited]
```

Return starter Node.

Returns

starter node with 00 NodeValue

Definition at line 167 of file model.h.

```
00167 { return starterNode;}
```

8.9.3.14 Train()

```
void Markov::API::MarkovPasswords::Train (
            const char * datasetFileName,
            char delimiter,
            int threads )
```

Train the model with the dataset file.

Parameters

datasetFileName	- Ifstream* to the dataset. If null, use class member
delimiter	- a character, same as the delimiter in dataset content
threads	- number of OS threads to spawn

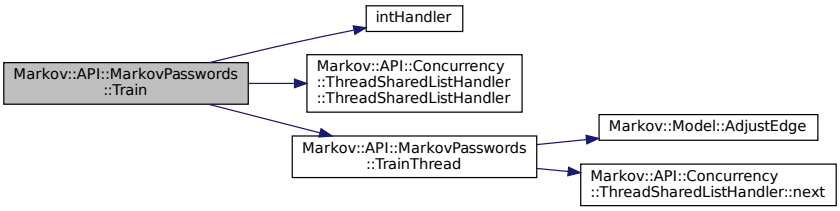
```
Markov::API::MarkovPasswords mp;
mp.Import("models/2gram.mdl");
mp.Train("password.corpus");
```

Definition at line 57 of file markovPasswords.cpp.

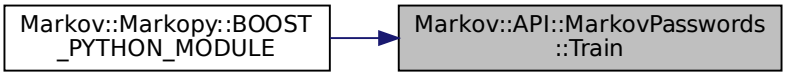
```
00057                                                                                                {
00058   signal(SIGINT, intHandler);
00059     Markov::API::Concurrency::ThreadSharedListHandler listhandler(datasetFileName);
00060     auto start = std::chrono::high_resolution_clock::now();
00061
00062     std::vector<std::thread*> threadsV;
00063     for(int i=0;i<threads;i++){
00064         threadsV.push_back(new std::thread(&Markov::API::MarkovPasswords::TrainThread, this,
       &listhandler, delimiter));
00065     }
00066
00067     for(int i=0;i<threads;i++){
00068         threadsV[i]->join();
00069         delete threadsV[i];
00070     }
00071     auto finish = std::chrono::high_resolution_clock::now();
00072     std::chrono::duration<double> elapsed = finish - start;
00073     std::cout << "Elapsed time: " << elapsed.count() << " s\n";
00074
00075 }
```

References intHandler(), Markov::API::Concurrency::ThreadSharedListHandler::ThreadSharedListHandler(), and

TrainThread().
Referenced by Markov::Markopy::BOOST_PYTHON_MODULE().
Here is the call graph for this function:



Here is the caller graph for this function:



8.9.3.15 TrainThread()

```
void Markov::API::MarkovPasswords::TrainThread (
            Markov::API::Concurrency::ThreadSharedListHandler * listhandler,
            char delimiter ) [private]
```

A single thread invoked by the Train function.

Parameters

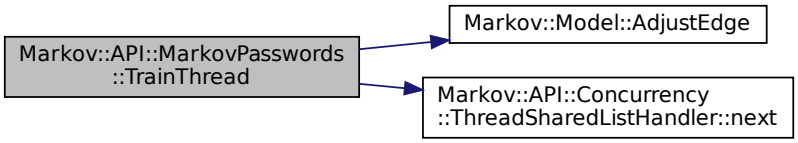
listhandler	- Listhandler class to read corpus from
delimiter	- a character, same as the delimiter in dataset content

Definition at line 77 of file markovPasswords.cpp.

```
00077                {
00078     char format_str[] ="%ld,%s";
00079     format_str[2]=delimiter;
00080     std::string line;
00081     while (listhandler->next(&line) && keepRunning) {
00082         long int oc;
00083         if (line.size() > 100) {
00084             line = line.substr(0, 100);
00085         }
00086         char* linebuf = new char[line.length()+5];
00087 #ifdef _WIN32
00088         sscanf_s(line.c_str(), "%ld,%s", &oc, linebuf, line.length()+5); //<== changed format_str to-> "%ld,%s"
00089 #else
00090         sscanf(line.c_str(), format_str, &oc, linebuf);
00091 #endif
00092         this->AdjustEdge((const char*)linebuf, oc);
00093         delete linebuf;
00094     }
00095 }
```

References Markov::Model< NodeStorageType >::AdjustEdge(), keepRunning, and Markov::API::Concurrency::ThreadSharedListHa
Referenced by Train().

Here is the call graph for this function:



Here is the caller graph for this function:

8.9.4 Member Data Documentation

8.9.4.1 datasetFile

`std::ifstream* Markov::API::MarkovPasswords::datasetFile [private]`
Definition at line 106 of file markovPasswords.h.

8.9.4.2 edges

`std::vector<Edge<char >*> Markov::Model< char >::edges [private], [inherited]`
A list of all edges in this model.
Definition at line 195 of file model.h.

8.9.4.3 modelSavefile

`std::ofstream* Markov::API::MarkovPasswords::modelSavefile [private]`
Dataset file input of our system

Definition at line 107 of file markovPasswords.h.

8.9.4.4 nodes

`std::map<char , Node<char >*> Markov::Model< char >::nodes [private], [inherited]`
Map LeftNode is the Nodes NodeValue Map RightNode is the node pointer.
Definition at line 184 of file model.h.

8.9.4.5 outputFile

`std::ofstream* Markov::API::MarkovPasswords::outputFile [private]`
File to save model of our system

Definition at line 108 of file markovPasswords.h.

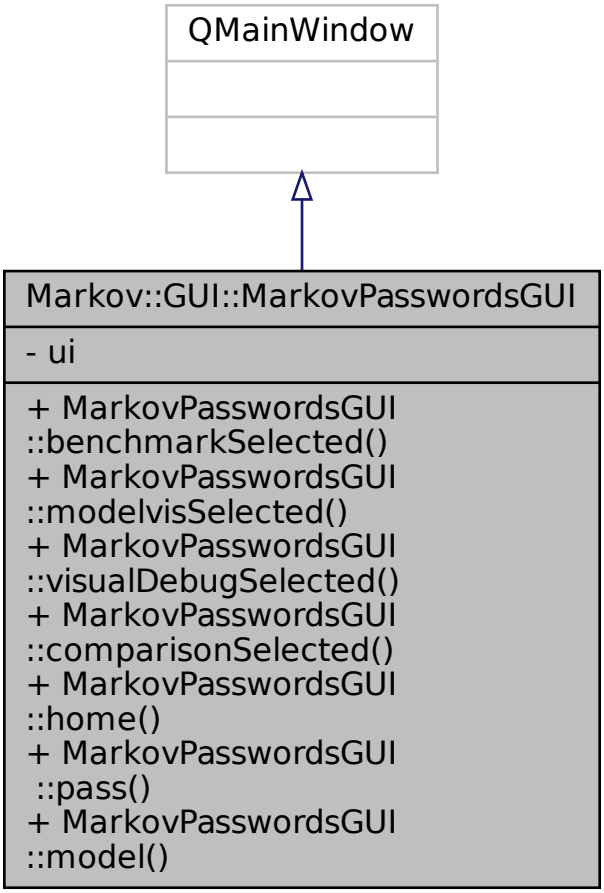
8.9.4.6 starterNode

`Node<char >* Markov::Model< char >::starterNode [private], [inherited]`
Starter Node of this model.
Definition at line 189 of file model.h.
The documentation for this class was generated from the following files:

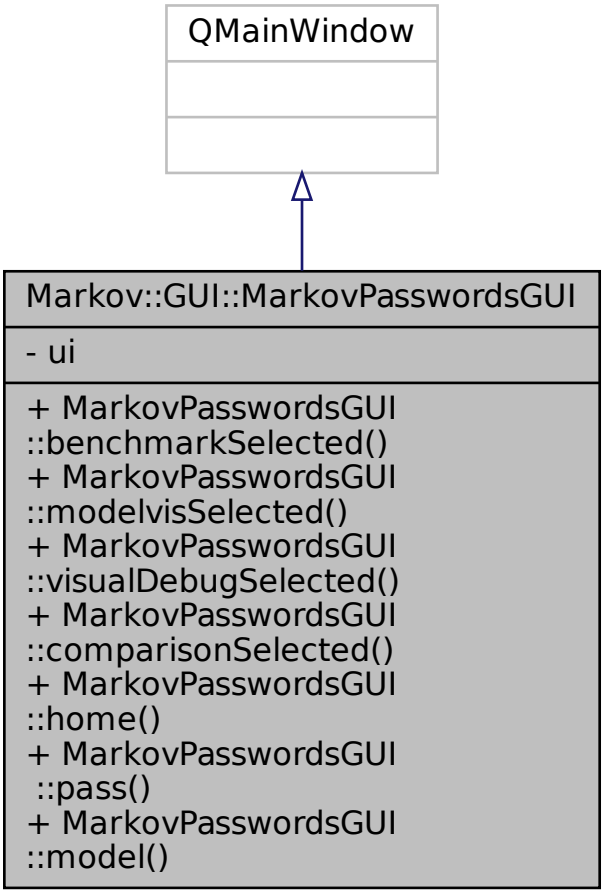
- markovPasswords.h
- markovPasswords.cpp

8.10 Markov::GUI::MarkovPasswordsGUI Class Reference

Reporting UI.
`#include <MarkovPasswordsGUI.h>`
Inheritance diagram for Markov::GUI::MarkovPasswordsGUI:



Collaboration diagram for Markov::GUI::MarkovPasswordsGUI:



Public Slots

- void MarkovPasswordsGUI::benchmarkSelected ()
- void MarkovPasswordsGUI::modelvisSelected ()
- void MarkovPasswordsGUI::visualDebugSelected ()
- void MarkovPasswordsGUI::comparisonSelected ()
- void MarkovPasswordsGUI::home ()
- void MarkovPasswordsGUI ::pass ()
- void MarkovPasswordsGUI::model ()

Private Attributes

- Ui::MarkovPasswordsGUIClass ui

8.10.1 Detailed Description

Reporting UI.
UI for reporting and debugging tools for MarkovPassword
Definition at line 13 of file MarkovPasswordsGUI.h.

8.10.2 Member Function Documentation

8.10.2.1 MarkovPasswordsGUI ::pass

```
void Markov::GUI::MarkovPasswordsGUI::MarkovPasswordsGUI ::pass ( )  [slot]
```

8.10.2.2 MarkovPasswordsGUI::benchmarkSelected

```
void Markov::GUI::MarkovPasswordsGUI::MarkovPasswordsGUI::benchmarkSelected ( )  [slot]
```

8.10.2.3 MarkovPasswordsGUI::comparisonSelected

```
void Markov::GUI::MarkovPasswordsGUI::MarkovPasswordsGUI::comparisonSelected ( )  [slot]
```

8.10.2.4 MarkovPasswordsGUI::home

```
void Markov::GUI::MarkovPasswordsGUI::MarkovPasswordsGUI::home ( )  [slot]
```

8.10.2.5 MarkovPasswordsGUI::model

```
void Markov::GUI::MarkovPasswordsGUI::MarkovPasswordsGUI::model ( )  [slot]
```

8.10.2.6 MarkovPasswordsGUI::modelvisSelected

```
void Markov::GUI::MarkovPasswordsGUI::MarkovPasswordsGUI::modelvisSelected ( )  [slot]
```

8.10.2.7 MarkovPasswordsGUI::visualDebugSelected

```
void Markov::GUI::MarkovPasswordsGUI::MarkovPasswordsGUI::visualDebugSelected ( )  [slot]
```

8.10.3 Member Data Documentation

8.10.3.1 ui

```
Ui::MarkovPasswordsGUIClass Markov::GUI::MarkovPasswordsGUI::ui  [private]
```

Definition at line 17 of file MarkovPasswordsGUI.h.

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

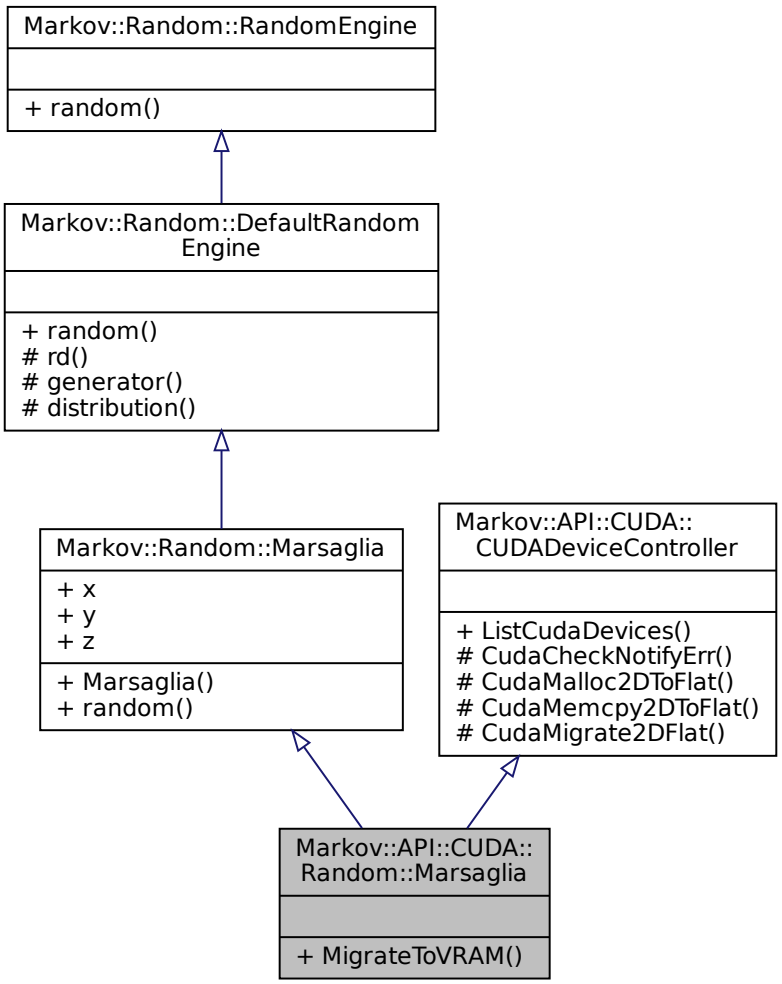
- MarkovPasswordsGUI.h

8.11 Markov::API::CUDA::Random::Marsaglia Class Reference

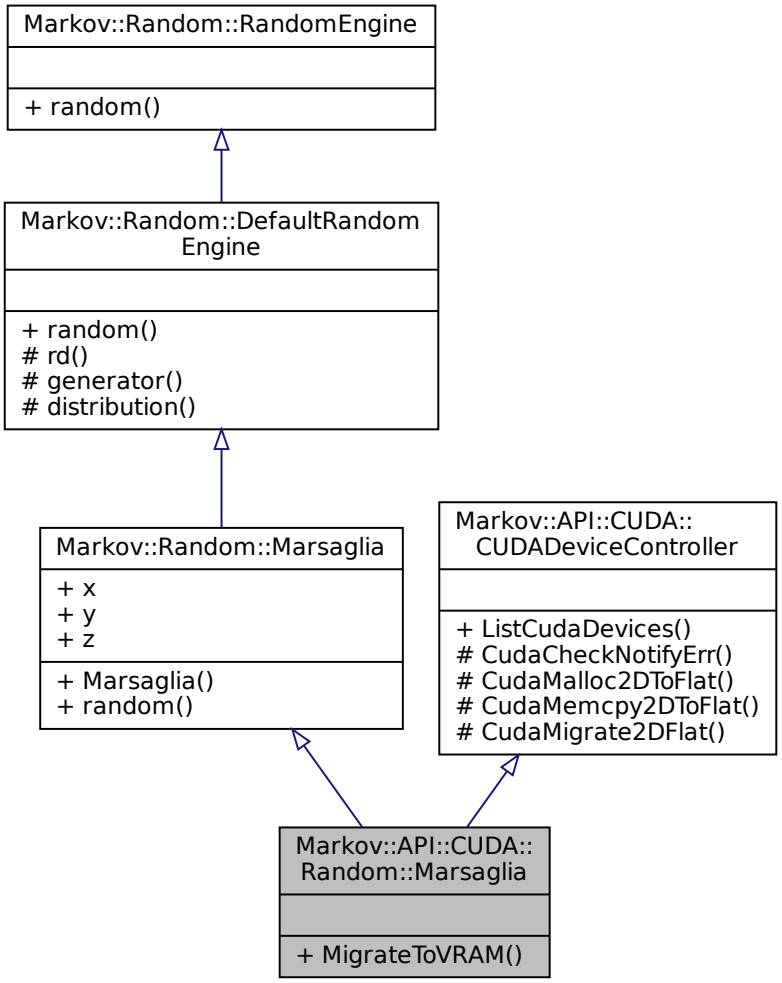
Extension of Markov::Random::Marsaglia which is capable o working on **device** space.

```
#include <cudarandom.h>
```

Inheritance diagram for Markov::API::CUDA::Random::Marsaglia:



Collaboration diagram for Markov::API::CUDA::Random::Marsaglia:



Public Member Functions

- unsigned long random ()

  Generate Random Number.

Static Public Member Functions

- static unsigned long * MigrateToVRAM (Markov::API::CUDA::Random::Marsaglia *MEarr, long int gridSize)

  Migrate a Marsaglia[] to VRAM as seedChunk.
- static __host__ void ListCudaDevices ()

  List CUDA devices in the system.

Public Attributes

- unsigned long x
- unsigned long y
- unsigned long z

Protected Member Functions

- std::random_device & rd ()

  Default random device for seeding.
- std::default_random_engine & generator ()

  Default random engine for seeding.
- std::uniform_int_distribution< long long unsigned > & distribution ()

  Distribution schema for seeding.

Static Protected Member Functions

- static __host__ int CudaCheckNotifyErr (cudaError_t _status, const char *msg, bool bExit=true)

  Check results of the last operation on GPU.
- template<typename T >
  static __host__ cudaError_t CudaMalloc2DToFlat (T **dst, int row, int col)

  Malloc a 2D array in device space.
- template<typename T >
  static __host__ cudaError_t CudaMemcpy2DToFlat (T *dst, T **src, int row, int col)

  Memcpy a 2D array in device space after flattening.
- template<typename T >
  static __host__ cudaError_t CudaMigrate2DFlat (T **dst, T **src, int row, int col)

  Both malloc and memcpy a 2D array into device VRAM.

8.11.1 Detailed Description

Extension of Markov::Random::Marsaglia which is capable o working on **device** space.
Definition at line 11 of file cudarandom.h.

8.11.2 Member Function Documentation

8.11.2.1 CudaCheckNotifyErr()

```
static __host__ int Markov::API::CUDA::CUDADeviceController::CudaCheckNotifyErr (
            cudaError_t _status,
            const char * msg,
            bool bExit = true )  [static], [protected], [inherited]
```

Check results of the last operation on GPU.
Check the status returned from cudaMalloc/cudaMemcpy to find failures.
If a failure occurs, its assumed beyond redemption, and exited.

Parameters

_status	Cuda error status to check
msg	Message to print in case of a failure

Returns

0 if successful, 1 if failure. **Example output:**

```
char *da, a = "test";
cudastatus = cudaMalloc((char **)&da, 5*sizeof(char*));
CudaCheckNotifyErr(cudastatus, "Failed to allocate VRAM for *da.\n");
```

8.11.2.2 CudaMalloc2DToFlat()

```
template<typename T >
```

```
static __host__ cudaError_t Markov::API::CUDA::CUDADeviceController::CudaMalloc2DToFlat (
            T ** dst,
            int row,
            int col )  [inline], [static], [protected], [inherited]
```

Malloc a 2D array in device space.
This function will allocate enough space on VRAM for flattened 2D array.

Parameters

dst	destination pointer
row	row size of the 2d array
col	column size of the 2d array

Returns

cudaError_t status of the cudaMalloc operation

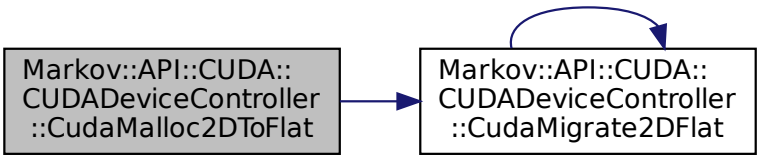
Example output:

```
cudaError_t cudastatus;
char* dst;
cudastatus = CudaMalloc2DToFlat<char>(&dst, 5, 15);
if(cudastatus!=cudaSuccess){
    CudaCheckNotifyErr(cudastatus, "  CudaMalloc2DToFlat Failed.", false);
}
```

Definition at line 73 of file cudaDeviceController.h.

```
00073                                                                    {
00074             cudaError_t cudastatus = cudaMalloc((T **)dst, row*col*sizeof(T));
00075             CudaCheckNotifyErr(cudastatus, "cudaMalloc Failed.", false);
00076             return cudastatus;
00077         }
```

References Markov::API::CUDA::CUDADeviceController::CudaMigrate2DFlat().
Here is the call graph for this function:



8.11.2.3 CudaMemcpy2DToFlat()

```
template<typename T >
static __host__ cudaError_t Markov::API::CUDA::CUDADeviceController::CudaMemcpy2DToFlat (
            T * dst,
            T ** src,
            int row,
            int col )  [inline], [static], [protected], [inherited]
```

Memcpy a 2D array in device space after flattening.
Resulting buffer will not be true 2D array.

Parameters

dst	destination pointer
rc	source pointer

Parameters

row	row size of the 2d array
col	column size of the 2d array

Returns

cudaError_t status of the cudaMalloc operation

Example output:

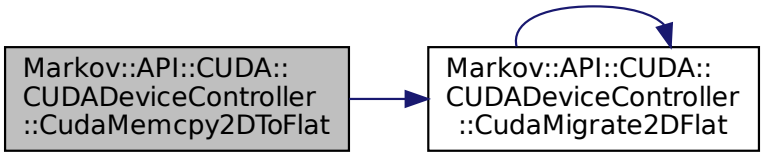
```
cudaError_t cudastatus;
char* dst;
cudastatus = CudaMalloc2DToFlat<char>(&dst, 5, 15);
CudaCheckNotifyErr(cudastatus, "  CudaMalloc2DToFlat Failed.", false);
cudastatus = CudaMemcpy2DToFlat<char>(*dst,src,15,15);
CudaCheckNotifyErr(cudastatus, "  CudaMemcpy2DToFlat Failed.", false);
```

Definition at line 101 of file cudaDeviceController.h.

```
00101                                                                                {
00102             T* tempbuf = new T[row*col];
00103             for(int i=0;i<row;i++){
00104                 memcpy(&(tempbuf[row*i]), src[i], col);
00105             }
00106             return cudaMemcpy(dst, tempbuf, row*col*sizeof(T), cudaMemcpyHostToDevice);
00107
00108         }
```

References Markov::API::CUDA::CUDADeviceController::CudaMigrate2DFlat().

Here is the call graph for this function:



8.11.2.4 CudaMigrate2DFlat()

```
template<typename T >
static __host__ cudaError_t Markov::API::CUDA::CUDADeviceController::CudaMigrate2DFlat (
             T ** dst,
             T ** src,
             int row,
             int col )  [inline], [static], [protected], [inherited]
```

Both malloc and memcpy a 2D array into device VRAM.

Resulting buffer will not be true 2D array.

Parameters

dst	destination pointer
rc	source pointer
row	row size of the 2d array
col	column size of the 2d array

Returns

cudaError_t status of the cudaMalloc operation

Example output:

```
cudaError_t cudastatus;
char* dst;
cudastatus = CudaMigrate2DFlat<long int>(
   &dst, this->valueMatrix, this->matrixSize, this->matrixSize);
CudaCheckNotifyErr(cudastatus, "    Cuda failed to initialize value matrix row.");
```

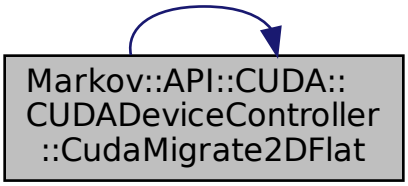
Definition at line 130 of file cudaDeviceController.h.

```
00130                                                                                                {
00131             cudaError_t cudastatus;
00132             cudastatus = CudaMalloc2DToFlat<T>(dst, row, col);
00133             if(cudastatus!=cudaSuccess){
00134                 CudaCheckNotifyErr(cudastatus, "  CudaMalloc2DToFlat Failed.", false);
00135                 return cudastatus;
00136             }
00137             cudastatus = CudaMemcpy2DToFlat<T>(*dst,src,row,col);
00138             CudaCheckNotifyErr(cudastatus, "  CudaMemcpy2DToFlat Failed.", false);
00139             return cudastatus;
00140         }
```

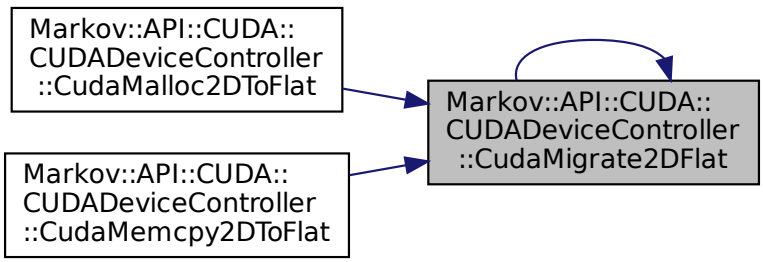
References Markov::API::CUDA::CUDADeviceController::CudaMigrate2DFlat().

Referenced by Markov::API::CUDA::CUDADeviceController::CudaMalloc2DToFlat(), Markov::API::CUDA::CUDADeviceController::Cu and Markov::API::CUDA::CUDADeviceController::CudaMigrate2DFlat().

Here is the call graph for this function:



Here is the caller graph for this function:



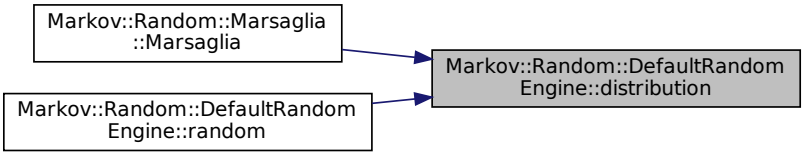
8.11.2.5 distribution()

```
std::uniform_int_distribution<long long unsigned>& Markov::Random::DefaultRandomEngine::distribution
( )  [inline], [protected], [inherited]
```

Distribution schema for seeding.
Definition at line 81 of file random.h.

```
00081                                                                   {
00082             static std::uniform_int_distribution<long long unsigned> _distribution(0, 0xffffFFFF);
00083             return _distribution;
00084         }
```

Referenced by Markov::Random::Marsaglia::Marsaglia(), and Markov::Random::DefaultRandomEngine::random().
Here is the caller graph for this function:



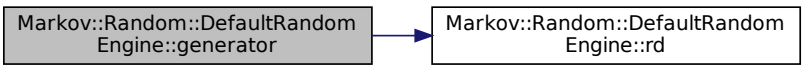
8.11.2.6 generator()

`std::default_random_engine& Markov::Random::DefaultRandomEngine::generator ( ) [inline], [protected], [inherited]`

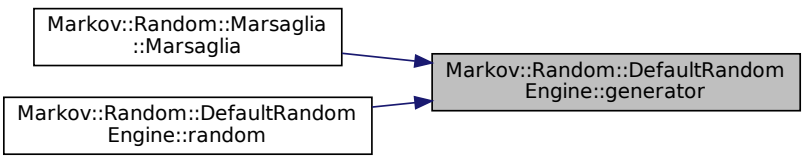
Default random engine for seeding.
Definition at line 73 of file random.h.

```
00073                                                 {
00074             static std::default_random_engine _generator(rd()());
00075             return _generator;
00076         }
```

References Markov::Random::DefaultRandomEngine::rd().
Referenced by Markov::Random::Marsaglia::Marsaglia(), and Markov::Random::DefaultRandomEngine::random().
Here is the call graph for this function:



Here is the caller graph for this function:



8.11.2.7 ListCudaDevices()

`static __host__ void Markov::API::CUDA::CUDADeviceController::ListCudaDevices ( ) [static],`

[inherited]
List CUDA devices in the system.
This function will print details of every CUDA capable device in the system.
Example output:

```
Device Number: 0
Device name: GeForce RTX 2070
Memory Clock Rate (KHz): 7001000
Memory Bus Width (bits): 256
Peak Memory Bandwidth (GB/s): 448.064
Max Linear Threads: 1024
```

8.11.2.8 MigrateToVRAM()

```
static unsigned long* Markov::API::CUDA::Random::Marsaglia::MigrateToVRAM (
            Markov::API::CUDA::Random::Marsaglia * MEarr,
            long int gridSize )  [inline], [static]
```

Migrate a Marsaglia[] to VRAM as seedChunk.

Parameters

MEarr	Array of Marsaglia Engines
gridSize	GridSize of the CUDA Kernel, aka size of array

Returns

pointer to the resulting seed chunk in device VRAM.

Definition at line 19 of file cudarandom.h.

```
00019                                                                                                                {
00020             cudaError_t cudastatus;
00021             unsigned long* seedChunk;
00022             cudastatus = cudaMalloc((unsigned long**)&seedChunk, gridSize*3*sizeof(unsigned long));
00023             CudaCheckNotifyErr(cudastatus, "Failed to allocate seed buffer");
00024             unsigned long *temp = new unsigned long[gridSize*3];
00025             for(int i=0;i<gridSize;i++){
00026                 temp[i*3]   = MEarr[i].x;
00027                 temp[i*3+1] = MEarr[i].y;
00028                 temp[i*3+2] = MEarr[i].z;
00029             }
00030             //for(int i=0;i<gridSize*3;i++) std::cout << temp[i] << "\n";
00031             cudaMemcpy(seedChunk, temp, gridSize*3*sizeof(unsigned long), cudaMemcpyHostToDevice);
00032             CudaCheckNotifyErr(cudastatus, "Failed to memcpy seed buffer.");
00033             return seedChunk;
00034         }
```

References Markov::Random::Marsaglia::x, Markov::Random::Marsaglia::y, and Markov::Random::Marsaglia::z.

8.11.2.9 random()

```
unsigned long Markov::Random::Marsaglia::random ( )  [inline], [virtual], [inherited]
```

Generate Random Number.

Returns

random number in long range.

Reimplemented from Markov::Random::DefaultRandomEngine.
Definition at line 131 of file random.h.

```
00131                                   {
00132         unsigned long t;
00133         x ^= x << 16;
00134         x ^= x >> 5;
00135         x ^= x << 1;
00136 
00137         t = x;
00138         x = y;
00139         y = z;
00140         z = t ^ x ^ y;
00141 
00142         return z;
```

```
00143     }
```

References Markov::Random::Marsaglia::x, Markov::Random::Marsaglia::y, and Markov::Random::Marsaglia::z.
Referenced by Markov::API::ModelMatrix::FastRandomWalkThread().
Here is the caller graph for this function:

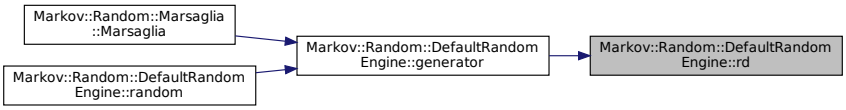
8.11.2.10 rd()

`std::random_device& Markov::Random::DefaultRandomEngine::rd ( )  [inline], [protected], [inherited]`

Default random device for seeding.
Definition at line 65 of file random.h.

```
00065                                   {
00066             static std::random_device _rd;
00067             return _rd;
00068         }
```

Referenced by Markov::Random::DefaultRandomEngine::generator().
Here is the caller graph for this function:



8.11.3 Member Data Documentation

8.11.3.1 x

`unsigned long Markov::Random::Marsaglia::x  [inherited]`

Definition at line 146 of file random.h.
Referenced by Markov::Random::Marsaglia::Marsaglia(), MigrateToVRAM(), and Markov::Random::Marsaglia::random().

8.11.3.2 y

`unsigned long Markov::Random::Marsaglia::y  [inherited]`

Definition at line 147 of file random.h.
Referenced by Markov::Random::Marsaglia::Marsaglia(), MigrateToVRAM(), and Markov::Random::Marsaglia::random().

8.11.3.3 z

`unsigned long Markov::Random::Marsaglia::z  [inherited]`

Definition at line 148 of file random.h.
Referenced by Markov::Random::Marsaglia::Marsaglia(), MigrateToVRAM(), and Markov::Random::Marsaglia::random().
The documentation for this class was generated from the following file:

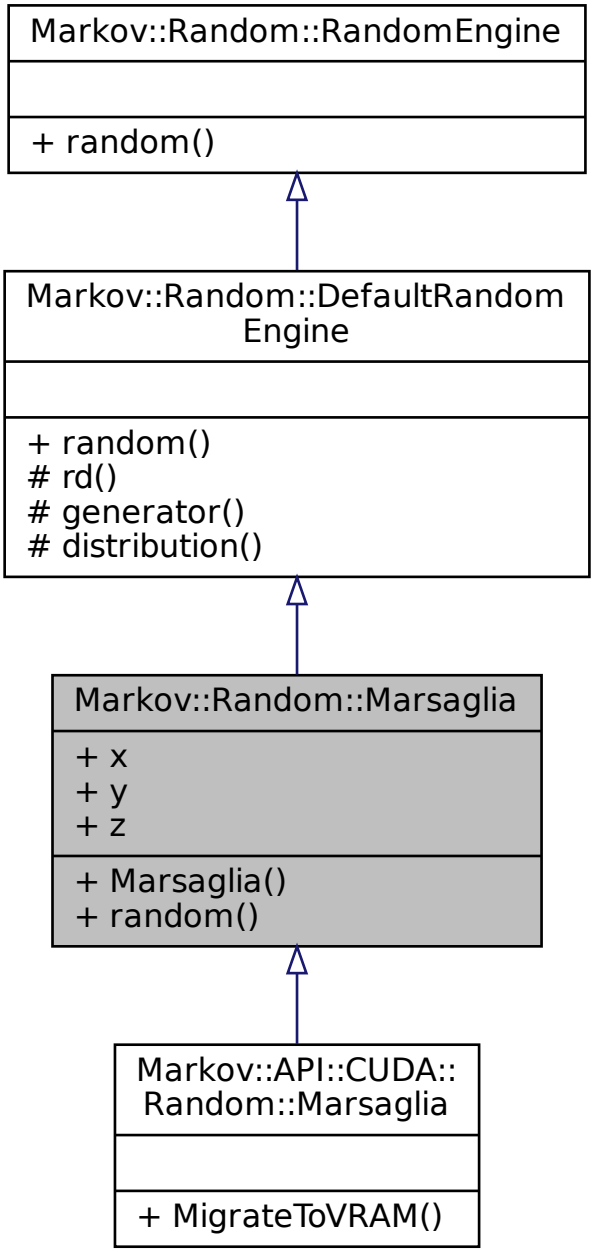
- cudarandom.h

8.12 Markov::Random::Marsaglia Class Reference

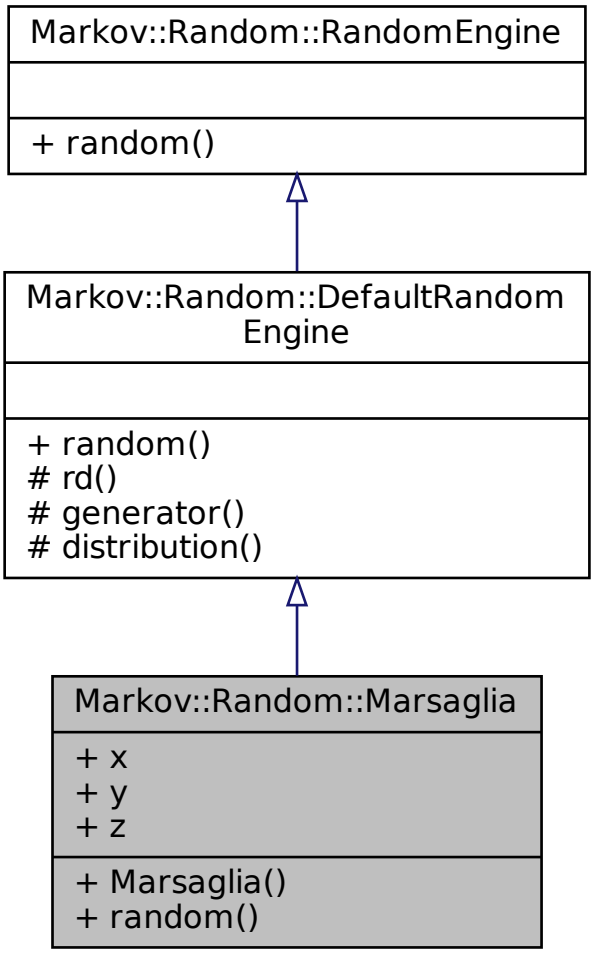
Implementation of Marsaglia Random Engine.

`#include <random.h>`

Inheritance diagram for Markov::Random::Marsaglia:



Collaboration diagram for Markov::Random::Marsaglia:



Public Member Functions

- Marsaglia ()

  Construct Marsaglia Engine.
- unsigned long random ()

  Generate Random Number.

Public Attributes

- unsigned long x
- unsigned long y
- unsigned long z

Protected Member Functions

- std::random_device & rd ()

  Default random device for seeding.

- std::default_random_engine & generator ()

  Default random engine for seeding.
- std::uniform_int_distribution< long long unsigned > & distribution ()

  Distribution schema for seeding.

8.12.1 Detailed Description

Implementation of Marsaglia Random Engine.
This is an implementation of Marsaglia Random engine, which for most use cases is a better fit than other solutions.
Very simple mathematical formula to generate pseudorandom integer, so its crazy fast.
This implementation of the Marsaglia Engine is seeded by random.h default random engine. RandomEngine is only seeded once so its not a performance issue.
Example Use: Using Marsaglia Engine with RandomWalk

```
Markov::Model<char> model;
Model.import("model.mdl");
char* res = new char[11];
Markov::Random::Marsaglia MarsagliaRandomEngine;
for (int i = 0; i < 10; i++) {
    this->RandomWalk(&MarsagliaRandomEngine, 5, 10, res);
    std::cout « res « "\n";
 }
```

Example Use: Generating a random number with Marsaglia Engine

```
Markov::Random::Marsaglia me;
std::cout « me.random();
```

Definition at line 116 of file random.h.

8.12.2 Constructor & Destructor Documentation

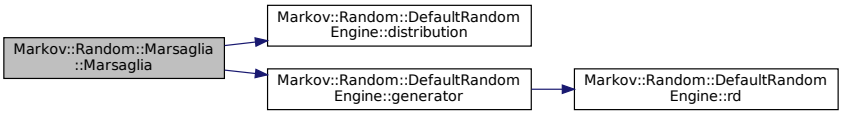
8.12.2.1 Marsaglia()

```
Markov::Random::Marsaglia::Marsaglia ( )  [inline]
```

Construct Marsaglia Engine.
Initialize x,y and z using the default random engine.
Definition at line 123 of file random.h.

```
00123                 {
00124             this->x = this->distribution()(this->generator());
00125             this->y = this->distribution()(this->generator());
00126             this->z = this->distribution()(this->generator());
00127             //std::cout « "x: " « x « ", y: " « y « ", z: " « z « "\n";
00128         }
```

References Markov::Random::DefaultRandomEngine::distribution(), Markov::Random::DefaultRandomEngine::generator(), x, y, and z.
Here is the call graph for this function:



8.12.3 Member Function Documentation

8.12.3.1 distribution()

```
std::uniform_int_distribution<long long unsigned>& Markov::Random::DefaultRandomEngine::distribution
( )  [inline], [protected], [inherited]
```

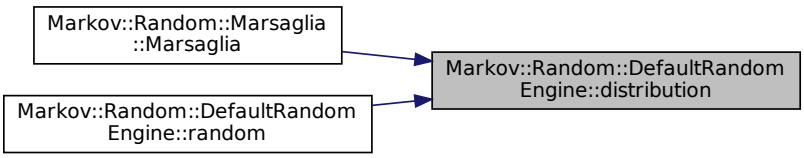
Distribution schema for seeding.

Definition at line 81 of file random.h.

```
00081                                                                   {
00082             static std::uniform_int_distribution<long long unsigned> _distribution(0, 0xffffFFFF);
00083             return _distribution;
00084         }
```

Referenced by Marsaglia(), and Markov::Random::DefaultRandomEngine::random().

Here is the caller graph for this function:



8.12.3.2 generator()

`std::default_random_engine& Markov::Random::DefaultRandomEngine::generator ( ) [inline], [protected], [inherited]`

Default random engine for seeding.

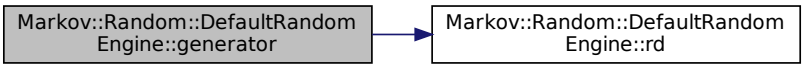
Definition at line 73 of file random.h.

```
00073                                                 {
00074             static std::default_random_engine _generator(rd()());
00075             return _generator;
00076         }
```

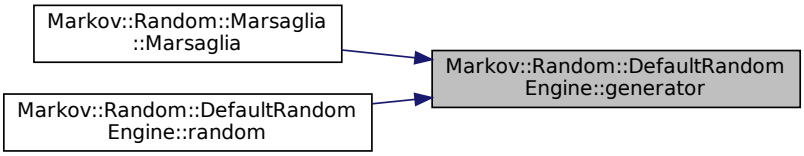
References Markov::Random::DefaultRandomEngine::rd().

Referenced by Marsaglia(), and Markov::Random::DefaultRandomEngine::random().

Here is the call graph for this function:



Here is the caller graph for this function:



8.12.3.3 random()

`unsigned long Markov::Random::Marsaglia::random ( ) [inline], [virtual]`

Generate Random Number.

Returns

random number in long range.

Reimplemented from Markov::Random::DefaultRandomEngine.
Definition at line 131 of file random.h.

```
00131                                   {
00132         unsigned long t;
00133         x ^= x « 16;
00134         x ^= x » 5;
00135         x ^= x « 1;
00136 
00137         t = x;
00138         x = y;
00139         y = z;
00140         z = t ^ x ^ y;
00141 
00142         return z;
00143     }
```

References x, y, and z.
Referenced by Markov::API::ModelMatrix::FastRandomWalkThread().
Here is the caller graph for this function:

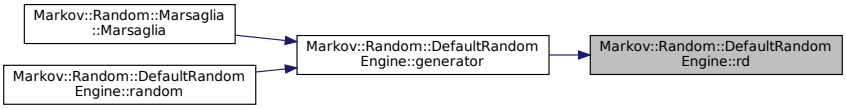
8.12.3.4 rd()

`std::random_device& Markov::Random::DefaultRandomEngine::rd ( )  [inline], [protected], [inherited]`

Default random device for seeding.
Definition at line 65 of file random.h.

```
00065                                   {
00066             static std::random_device _rd;
00067             return _rd;
00068         }
```

Referenced by Markov::Random::DefaultRandomEngine::generator().
Here is the caller graph for this function:



8.12.4 Member Data Documentation

8.12.4.1 x

`unsigned long Markov::Random::Marsaglia::x`

Definition at line 146 of file random.h.
Referenced by Marsaglia(), Markov::API::CUDA::Random::Marsaglia::MigrateToVRAM(), and random().

8.12.4.2 y

`unsigned long Markov::Random::Marsaglia::y`

Definition at line 147 of file random.h.

Referenced by Marsaglia(), Markov::API::CUDA::Random::Marsaglia::MigrateToVRAM(), and random().

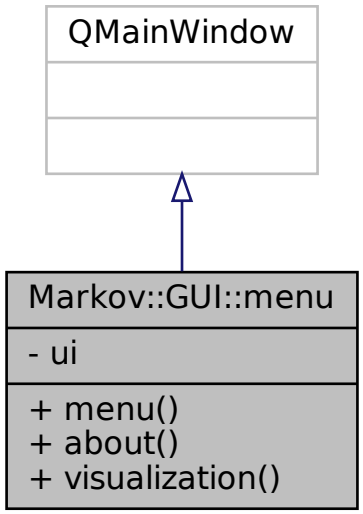
8.12.4.3 z

`unsigned long Markov::Random::Marsaglia::z`
Definition at line 148 of file random.h.
Referenced by Marsaglia(), Markov::API::CUDA::Random::Marsaglia::MigrateToVRAM(), and random().
The documentation for this class was generated from the following file:

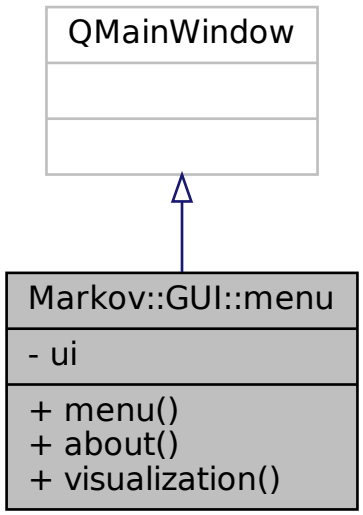
- random.h

8.13 Markov::GUI::menu Class Reference

QT Menu class.
`#include <menu.h>`
Inheritance diagram for Markov::GUI::menu:



Collaboration diagram for Markov::GUI::menu:



Public Slots

- void about ()
- void visualization ()

Public Member Functions

- menu (QWidget ∗parent=Q_NULLPTR)

Private Attributes

- Ui::main ui

8.13.1 Detailed Description

QT Menu class.
Definition at line 9 of file menu.h.

8.13.2 Constructor & Destructor Documentation

8.13.2.1 menu()

```
menu::menu (
            QWidget * parent = Q_NULLPTR )
```

Definition at line 8 of file menu.cpp.

```
00009     : QMainWindow(parent)
00010 {
00011     ui.setupUi(this);
00012
00013
00014     //QObject::connect(ui.pushButton, &QPushButton::clicked, this, [this] {about(); });
00015     QObject::connect(ui.visu, &QPushButton::clicked, this, [this] {visualization(); });
00016 }
```

8.13.3 Member Function Documentation

8.13.3.1 about

```
void menu::about ( )  [slot]
```

Definition at line 17 of file menu.cpp.

```
00017                 {
00018
00019
00020 }
```

8.13.3.2 visualization

```
void menu::visualization ( )  [slot]
```

Definition at line 21 of file menu.cpp.

```
00021                         {
00022     MarkovPasswordsGUI* w = new MarkovPasswordsGUI;
00023     w->show();
00024     this->close();
00025 }
```

8.13.4 Member Data Documentation

8.13.4.1 ui

```
Ui::main Markov::GUI::menu::ui  [private]
```

Definition at line 15 of file menu.h.

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

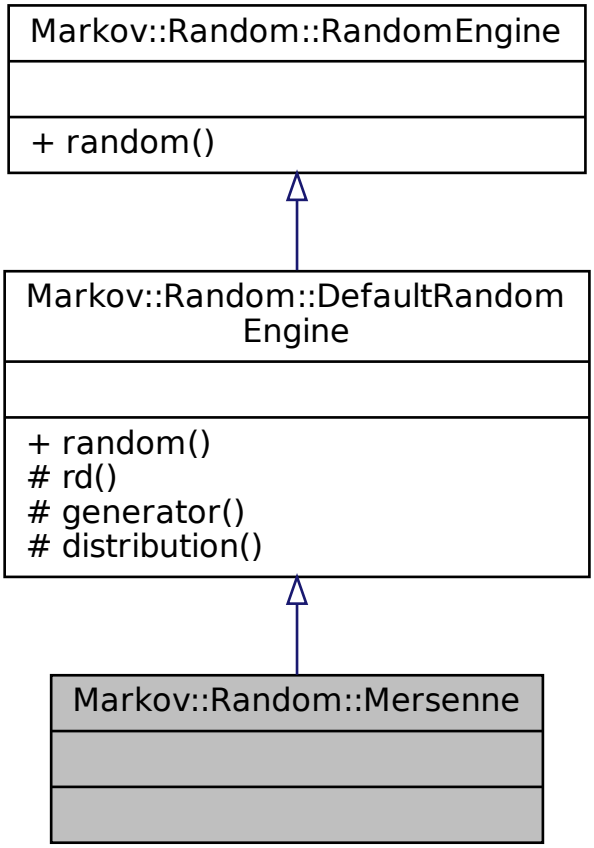
- menu.h
- menu.cpp

8.14 Markov::Random::Mersenne Class Reference

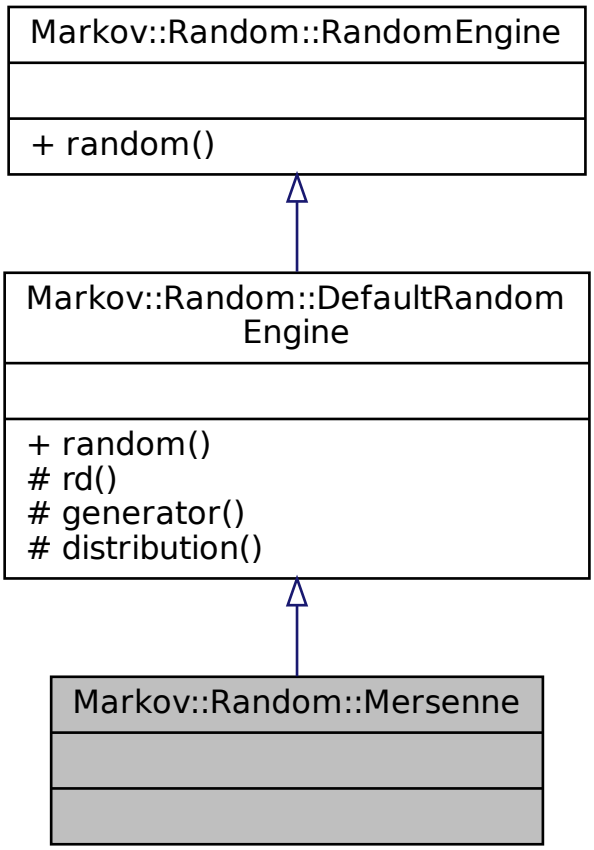
Implementation of Mersenne Twister Engine.

```
#include <random.h>
```

Inheritance diagram for Markov::Random::Mersenne:



Collaboration diagram for Markov::Random::Mersenne:



Public Member Functions

- unsigned long random ()

  Generate Random Number.

Protected Member Functions

- std::random_device & rd ()

  Default random device for seeding.
- std::default_random_engine & generator ()

  Default random engine for seeding.
- std::uniform_int_distribution< long long unsigned > & distribution ()

  Distribution schema for seeding.

8.14.1 Detailed Description

Implementation of Mersenne Twister Engine.
This is an implementation of Mersenne Twister Engine, which is slow but is a good implementation for high entropy pseudorandom.
Example Use: Using Mersenne Engine with RandomWalk

```
Markov::Model<char> model;
Model.import("model.mdl");
char* res = new char[11];
Markov::Random::Mersenne MersenneTwisterEngine;
for (int i = 0; i < 10; i++) {
    this->RandomWalk(&MersenneTwisterEngine, 5, 10, res);
    std::cout « res « "\n";
 }
```

Example Use: Generating a random number with Marsaglia Engine

```
Markov::Random::Mersenne me;
std::cout « me.random();
```

Definition at line 176 of file random.h.

8.14.2 Member Function Documentation

8.14.2.1 distribution()

`std::uniform_int_distribution<long long unsigned>& Markov::Random::DefaultRandomEngine::distribution ( ) [inline], [protected], [inherited]`

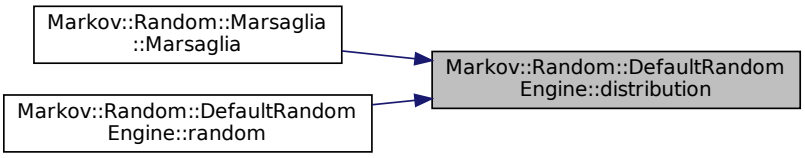
Distribution schema for seeding.

Definition at line 81 of file random.h.

```
00081                                                                    {
00082             static std::uniform_int_distribution<long long unsigned> _distribution(0, 0xffffFFFF);
00083             return _distribution;
00084         }
```

Referenced by Markov::Random::Marsaglia::Marsaglia(), and Markov::Random::DefaultRandomEngine::random().

Here is the caller graph for this function:



8.14.2.2 generator()

`std::default_random_engine& Markov::Random::DefaultRandomEngine::generator ( ) [inline], [protected], [inherited]`

Default random engine for seeding.

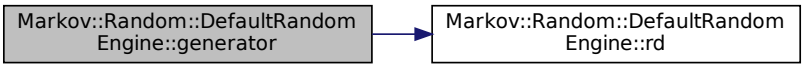
Definition at line 73 of file random.h.

```
00073                                                {
00074             static std::default_random_engine _generator(rd()());
00075             return _generator;
00076         }
```

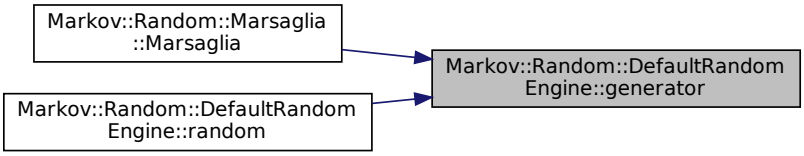
References Markov::Random::DefaultRandomEngine::rd().

Referenced by Markov::Random::Marsaglia::Marsaglia(), and Markov::Random::DefaultRandomEngine::random().

Here is the call graph for this function:



Here is the caller graph for this function:



8.14.2.3 random()

`unsigned long Markov::Random::DefaultRandomEngine::random ( ) [inline], [virtual], [inherited]`

Generate Random Number.

Returns

random number in long range.

Implements Markov::Random::RandomEngine.

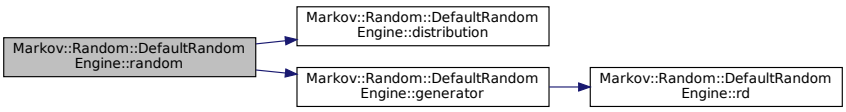
Reimplemented in Markov::Random::Marsaglia.

Definition at line 57 of file random.h.

```
00057                                   {
00058             return this->distribution()(this->generator());
00059         }
```

References Markov::Random::DefaultRandomEngine::distribution(), and Markov::Random::DefaultRandomEngine::generator().

Here is the call graph for this function:



8.14.2.4 rd()

`std::random_device& Markov::Random::DefaultRandomEngine::rd ( ) [inline], [protected], [inherited]`

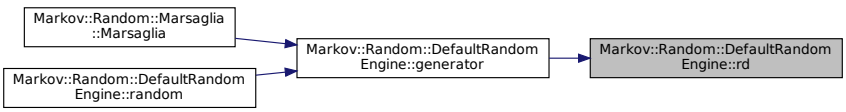
Default random device for seeding.

Definition at line 65 of file random.h.

```
00065                                    {
00066             static std::random_device _rd;
00067             return _rd;
00068         }
```

Referenced by Markov::Random::DefaultRandomEngine::generator().

Here is the caller graph for this function:

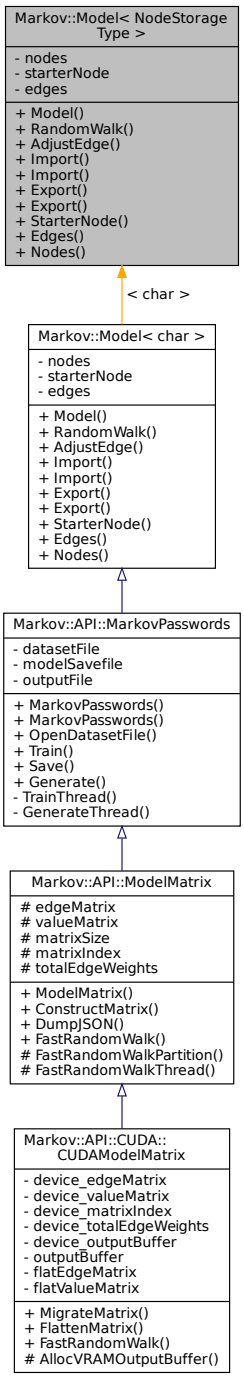


The documentation for this class was generated from the following file:

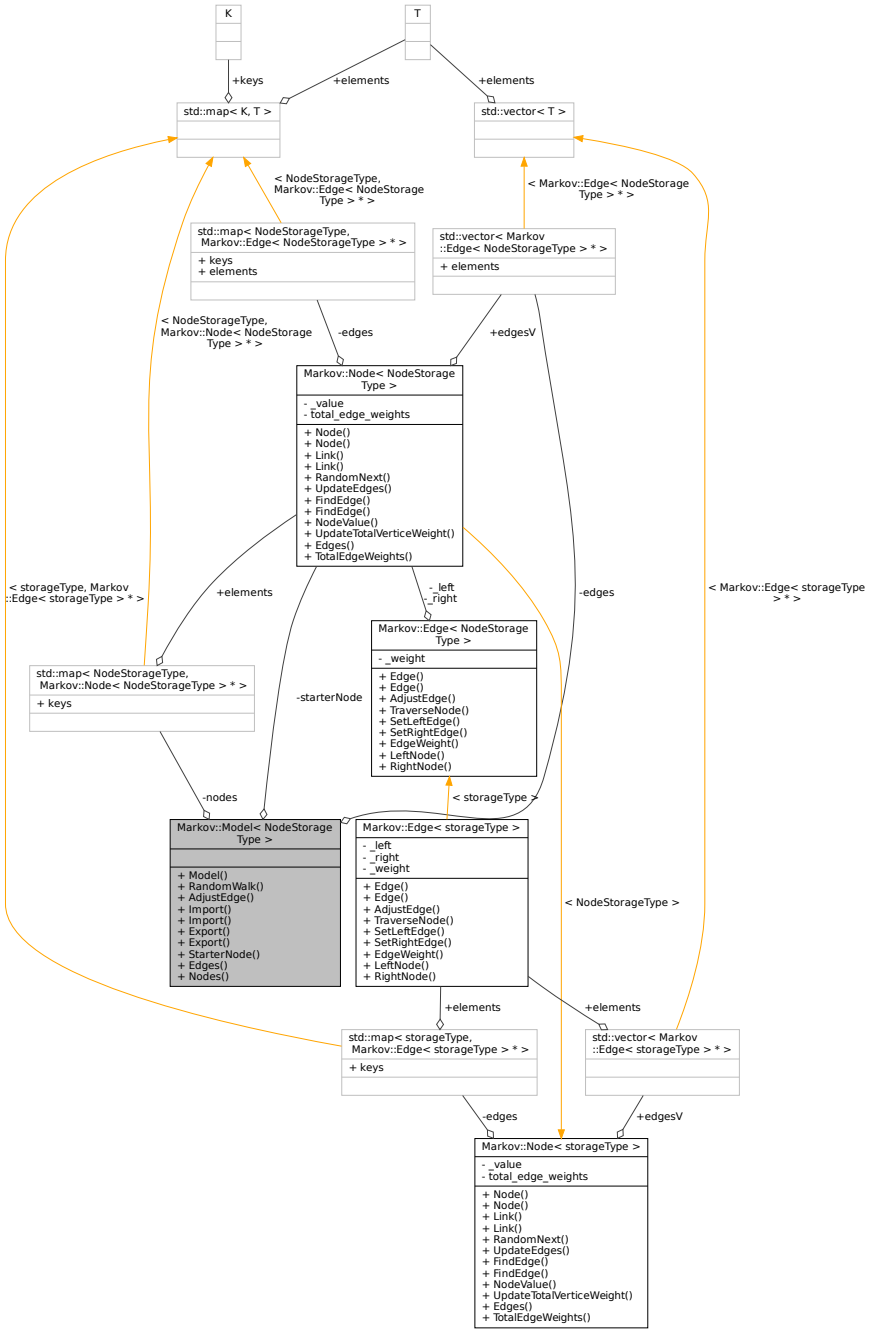
- random.h

8.15 Markov::Model< NodeStorageType > Class Template Reference

class for the final Markov Model, constructed from nodes and edges.
`#include <model.h>`
Inheritance diagram for Markov::Model< NodeStorageType >:



Collaboration diagram for Markov::Model< NodeStorageType >:



Public Member Functions

- Model ()

  Initialize a model with only start and end nodes.

- NodeStorageType * RandomWalk (Markov::Random::RandomEngine *randomEngine, int minSetting, int maxSetting, NodeStorageType *buffer)

  Do a random walk on this model.

- void AdjustEdge (const NodeStorageType *payload, long int occurrence)

  Adjust the model with a single string.

- bool Import (std::ifstream *)

Import a file to construct the model.

- bool Import (const char *filename)

  Open a file to import with filename, and call bool Model::Import with std::ifstream.
- bool Export (std::ofstream *)

  Export a file of the model.
- bool Export (const char *filename)

  Open a file to export with filename, and call bool Model::Export with std::ofstream.
- Node< NodeStorageType > * StarterNode ()

  Return starter Node.
- std::vector< Edge< NodeStorageType > * > * Edges ()

  Return a vector of all the edges in the model.
- std::map< NodeStorageType, Node< NodeStorageType > * > * Nodes ()

  Return starter Node.

Private Attributes

- std::map< NodeStorageType, Node< NodeStorageType > * > nodes

  Map LeftNode is the Nodes NodeValue Map RightNode is the node pointer.
- Node< NodeStorageType > * starterNode

  Starter Node of this model.
- std::vector< Edge< NodeStorageType > * > edges

  A list of all edges in this model.

8.15.1 Detailed Description

template<typename NodeStorageType>
class Markov::Model< NodeStorageType >

class for the final Markov Model, constructed from nodes and edges.
Each atomic piece of the generation result is stored in a node, while edges contain the relation weights. *Extending:* To extend the class, implement the template and inherit from it, as "class MyModel : public Markov::Model<char>". For a complete demonstration of how to extend the class, see MarkovPasswords.
Whole model can be defined as a list of the edges, as dangling nodes are pointless. This approach is used for the import/export operations. For more information on importing/exporting model, check out the github readme and wiki page.
Definition at line 41 of file model.h.

8.15.2 Constructor & Destructor Documentation

8.15.2.1 Model()

```
template<typename NodeStorageType >
Markov::Model< NodeStorageType >::Model
```

Initialize a model with only start and end nodes.
Initialize an empty model with only a starterNode Starter node is a special kind of node that has constant 0x00 value, and will be used to initiate the generation execution from.
Definition at line 201 of file model.h.

```
00201                                       {
00202     this->starterNode = new Markov::Node<NodeStorageType>(0);
00203     this->nodes.insert({ 0, this->starterNode });
00204 }
```

8.15.3 Member Function Documentation

8.15.3.1 AdjustEdge()

```
template<typename NodeStorageType >
void Markov::Model< NodeStorageType >::AdjustEdge (
            const NodeStorageType * payload,
            long int occurrence )
```

Adjust the model with a single string.
Start from the starter node, and for each character, AdjustEdge the edge EdgeWeight from current node to the next, until NULL character is reached.
Then, update the edge EdgeWeight from current node, to the terminator node.
This function is used for training purposes, as it can be used for adjusting the model with each line of the corpus file.
Example Use: Create an empty model and train it with string: "testdata"

```
Markov::Model<char> model;
char test[] = "testdata";
model.AdjustEdge(test, 15);
```

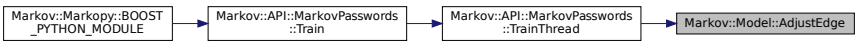
Parameters

string	- String that is passed from the training, and will be used to AdjustEdge the model with
occurrence	- Occurrence of this string.

Definition at line 323 of file model.h.

```
00323                                                                                                {
00324     NodeStorageType p = payload[0];
00325     Markov::Node<NodeStorageType>* curnode = this->starterNode;
00326     Markov::Edge<NodeStorageType>* e;
00327     int i = 0;
00328
00329     if (p == 0) return;
00330     while (p != 0) {
00331         e = curnode->FindEdge(p);
00332         if (e == NULL) return;
00333         e->AdjustEdge(occurrence);
00334         curnode = e->RightNode();
00335         p = payload[++i];
00336     }
00337
00338     e = curnode->FindEdge('\xff');
00339     e->AdjustEdge(occurrence);
00340     return;
00341 }
```

Referenced by Markov::API::MarkovPasswords::TrainThread().
Here is the caller graph for this function:



8.15.3.2 Edges()

```
template<typename NodeStorageType >
std::vector<Edge<NodeStorageType>*>* Markov::Model< NodeStorageType >::Edges ( )  [inline]
```

Return a vector of all the edges in the model.

Returns

vector of edges

Definition at line 172 of file model.h.

```
00172 { return &edges;}
```

8.15.3.3 Export() [1/2]

```
template<typename NodeStorageType >
bool Markov::Model< NodeStorageType >::Export (
            const char * filename )
```

Open a file to export with filename, and call bool Model::Export with std::ofstream.

Returns

True if successful, False for incomplete models or corrupt file formats

Example Use: Export file to filename

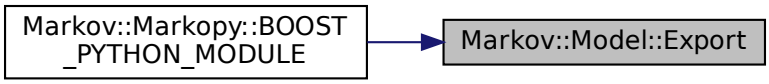
```
Markov::Model<char> model;
model.Export("test.mdl");
```

Definition at line 286 of file model.h.

```
00286                                                    {
00287     std::ofstream exportfile;
00288     exportfile.open(filename);
00289     return this->Export(&exportfile);
00290 }
```

Referenced by Markov::Markopy::BOOST_PYTHON_MODULE().

Here is the caller graph for this function:



8.15.3.4 Export() [2/2]

```
template<typename NodeStorageType >
bool Markov::Model< NodeStorageType >::Export (
            std::ofstream * f )
```

Export a file of the model.

File contains a list of edges. Format is: Left_repr;EdgeWeight;right_repr. For more information on the format, check out the project wiki or github readme.

Iterate over this vertices, and their edges, and write them to file.

Returns

True if successful, False for incomplete models.

Example Use: Export file to ofstream

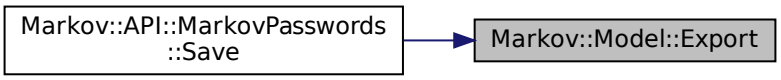
```
Markov::Model<char> model;
std::ofstream file("test.mdl");
model.Export(&file);
```

Definition at line 274 of file model.h.

```
00274                                                {
00275     Markov::Edge<NodeStorageType>* e;
00276     for (std::vector<int>::size_type i = 0; i != this->edges.size(); i++) {
00277         e = this->edges[i];
00278         //std::cout « e->LeftNode()->NodeValue() « "," « e->EdgeWeight() « "," «
      e->RightNode()->NodeValue() « "\n";
00279         *f « e->LeftNode()->NodeValue() « "," « e->EdgeWeight() « "," « e->RightNode()->NodeValue() «
      "\n";
00280     }
00281
00282     return true;
00283 }
```

Referenced by Markov::API::MarkovPasswords::Save().

Here is the caller graph for this function:



8.15.3.5 Import() [1/2]

```
template<typename NodeStorageType >
bool Markov::Model< NodeStorageType >::Import (
            const char * filename )
```

Open a file to import with filename, and call bool Model::Import with std::ifstream.

Returns

True if successful, False for incomplete models or corrupt file formats

Example Use: Import a file with filename

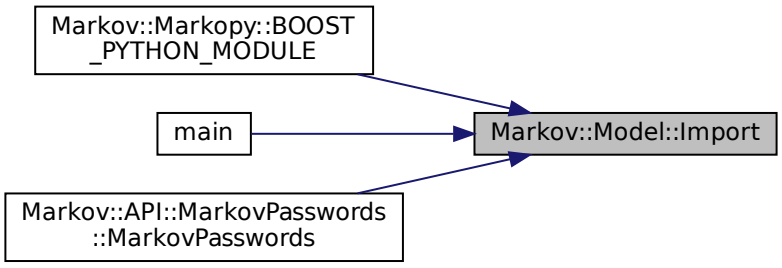
```
Markov::Model<char> model;
model.Import("test.mdl");
```

Definition at line 266 of file model.h.

```
00266                                                    {
00267     std::ifstream importfile;
00268     importfile.open(filename);
00269     return this->Import(&importfile);
00270
00271 }
```

Referenced by Markov::Markopy::BOOST_PYTHON_MODULE(), main(), and Markov::API::MarkovPasswords::MarkovPasswords().

Here is the caller graph for this function:



8.15.3.6 Import() [2/2]

```
template<typename NodeStorageType >
bool Markov::Model< NodeStorageType >::Import (
            std::ifstream * f )
```

Import a file to construct the model.

File contains a list of edges. For more info on the file format, check out the wiki and github readme pages. Format is: Left_repr;EdgeWeight;right_repr

Iterate over this list, and construct nodes and edges accordingly.

Returns

True if successful, False for incomplete models or corrupt file formats

Example Use: Import a file from ifstream

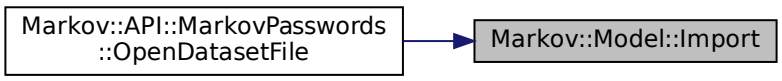
```
Markov::Model<char> model;
std::ifstream file("test.mdl");
model.Import(&file);
```

Definition at line 207 of file model.h.

```
00207                                                    {
00208     std::string cell;
00209
00210     char src;
00211     char target;
00212     long int oc;
00213
00214     while (std::getline(*f, cell)) {
00215         //std::cout « "cell: " « cell « std::endl;
00216         src = cell[0];
00217         target = cell[cell.length() - 1];
00218         char* j;
00219         oc = std::strtol(cell.substr(2, cell.length() - 2).c_str(),&j,10);
00220         //std::cout « oc « "\n";
00221         Markov::Node<NodeStorageType>* srcN;
00222         Markov::Node<NodeStorageType>* targetN;
00223         Markov::Edge<NodeStorageType>* e;
00224         if (this->nodes.find(src) == this->nodes.end()) {
00225             srcN = new Markov::Node<NodeStorageType>(src);
00226             this->nodes.insert(std::pair<char, Markov::Node<NodeStorageType>*>(src, srcN));
00227             //std::cout « "Creating new node at start.\n";
00228         }
00229         else {
00230             srcN = this->nodes.find(src)->second;
00231         }
00232
00233         if (this->nodes.find(target) == this->nodes.end()) {
00234             targetN = new Markov::Node<NodeStorageType>(target);
00235             this->nodes.insert(std::pair<char, Markov::Node<NodeStorageType>*>(target, targetN));
00236             //std::cout « "Creating new node at end.\n";
00237         }
00238         else {
00239             targetN = this->nodes.find(target)->second;
00240         }
00241         e = srcN->Link(targetN);
00242         e->AdjustEdge(oc);
00243         this->edges.push_back(e);
00244
00245         //std::cout « int(srcN->NodeValue()) « " --" « e->EdgeWeight() « "--> " «
    int(targetN->NodeValue()) « "\n";
00246
00247
00248     }
00249
00250     for (std::pair<unsigned char, Markov::Node<NodeStorageType>*> const& x : this->nodes) {
00251         //std::cout « "Total edges in EdgesV: " « x.second->edgesV.size() « "\n";
00252         std::sort (x.second->edgesV.begin(), x.second->edgesV.end(), [](Edge<NodeStorageType> *lhs,
    Edge<NodeStorageType> *rhs)->bool{
00253             return lhs->EdgeWeight() > rhs->EdgeWeight();
00254         });
00255         //for(int i=0;i<x.second->edgesV.size();i++)
00256         //  std::cout « x.second->edgesV[i]->EdgeWeight() « ", ";
00257         //std::cout « "\n";
00258     }
00259     //std::cout « "Total number of nodes: " « this->nodes.size() « std::endl;
00260     //std::cout « "Total number of edges: " « this->edges.size() « std::endl;
00261
00262     return true;
00263 }
```

Referenced by Markov::API::MarkovPasswords::OpenDatasetFile().

Here is the caller graph for this function:



8.15.3.7 Nodes()

```
template<typename NodeStorageType >
std::map<NodeStorageType, Node<NodeStorageType>*>* Markov::Model< NodeStorageType >::Nodes (
) [inline]
```

Return starter Node.

Returns

starter node with 00 NodeValue

Definition at line 177 of file model.h.

```
00177 { return &nodes;}
```

Referenced by Markov::API::ModelMatrix::ConstructMatrix().

Here is the caller graph for this function:

8.15.3.8 RandomWalk()

```
template<typename NodeStorageType >
NodeStorageType * Markov::Model< NodeStorageType >::RandomWalk (
            Markov::Random::RandomEngine * randomEngine,
            int minSetting,
            int maxSetting,
            NodeStorageType * buffer )
```

Do a random walk on this model.

Start from the starter node, on each node, invoke RandomNext using the random engine on current node, until terminator node is reached. If terminator node is reached before minimum length criateria is reached, ignore the last selection and re-invoke randomNext

If maximum length criteria is reached but final node is not, cut off the generation and proceed to the final node. This function takes Markov::Random::RandomEngine as a parameter to generate pseudo random numbers from

This library is shipped with two random engines, Marsaglia and Mersenne. While mersenne output is higher in entropy, most use cases don't really need super high entropy output, so Markov::Random::Marsaglia is preferable for better performance.

This function WILL NOT reallocate buffer. Make sure no out of bound writes are happening via maximum length criteria.

Example Use: Generate 10 lines, with 5 to 10 characters, and print the output. Use Marsaglia

```
Markov::Model<char> model;
Model.import("model.mdl");
char* res = new char[11];
Markov::Random::Marsaglia MarsagliaRandomEngine;
for (int i = 0; i < 10; i++) {
    this->RandomWalk(&MarsagliaRandomEngine, 5, 10, res);
    std::cout « res « "\n";
 }
```

Parameters

randomEngine	Random Engine to use for the random walks. For examples, see Markov::Random::Mersenne and Markov::Random::Marsaglia
minSetting	Minimum number of characters to generate

Parameters

maxSetting	Maximum number of character to generate
buffer	buffer to write the result to

Returns

Null terminated string that was generated.

Definition at line 293 of file model.h.

```
00293                                                   {
00294     Markov::Node<NodeStorageType>* n = this->starterNode;
00295     int len = 0;
00296     Markov::Node<NodeStorageType>* temp_node;
00297     while (true) {
00298         temp_node = n->RandomNext(randomEngine);
00299         if (len >= maxSetting) {
00300             break;
00301         }
00302         else if ((temp_node == NULL) && (len < minSetting)) {
00303             continue;
00304         }
00305
00306         else if (temp_node == NULL){
00307             break;
00308         }
00309
00310         n = temp_node;
00311
00312         buffer[len++] = n->NodeValue();
00313     }
00314
00315     //null terminate the string
00316     buffer[len] = 0x00;
00317
00318     //do something with the generated string
00319     return buffer; //for now
00320 }
```

Referenced by Markov::API::MarkovPasswords::GenerateThread().

Here is the caller graph for this function:

8.15.3.9 StarterNode()

```
template<typename NodeStorageType >
Node<NodeStorageType>* Markov::Model< NodeStorageType >::StarterNode ( )  [inline]
```

Return starter Node.

Returns

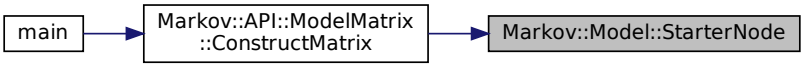
starter node with 00 NodeValue

Definition at line 167 of file model.h.

```
00167 { return starterNode;}
```

Referenced by Markov::API::ModelMatrix::ConstructMatrix().

Here is the caller graph for this function:



8.15.4 Member Data Documentation

8.15.4.1 edges

```
template<typename NodeStorageType >
std::vector<Edge<NodeStorageType>*> Markov::Model< NodeStorageType >::edges  [private]
```

A list of all edges in this model.
Definition at line 195 of file model.h.
Referenced by Markov::Model< char >::Edges().

8.15.4.2 nodes

```
template<typename NodeStorageType >
std::map<NodeStorageType, Node<NodeStorageType>*> Markov::Model< NodeStorageType >::nodes
[private]
```

Map LeftNode is the Nodes NodeValue Map RightNode is the node pointer.
Definition at line 184 of file model.h.
Referenced by Markov::Model< char >::Nodes().

8.15.4.3 starterNode

```
template<typename NodeStorageType >
Node<NodeStorageType>* Markov::Model< NodeStorageType >::starterNode  [private]
```

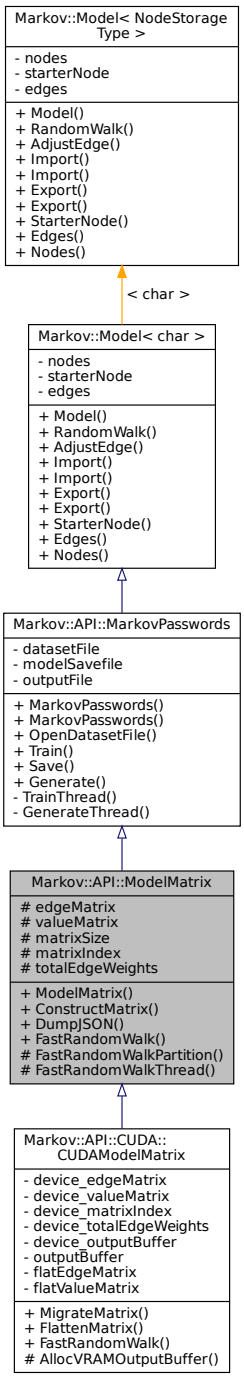
Starter Node of this model.
Definition at line 189 of file model.h.
Referenced by Markov::Model< char >::StarterNode().
The documentation for this class was generated from the following file:

- model.h

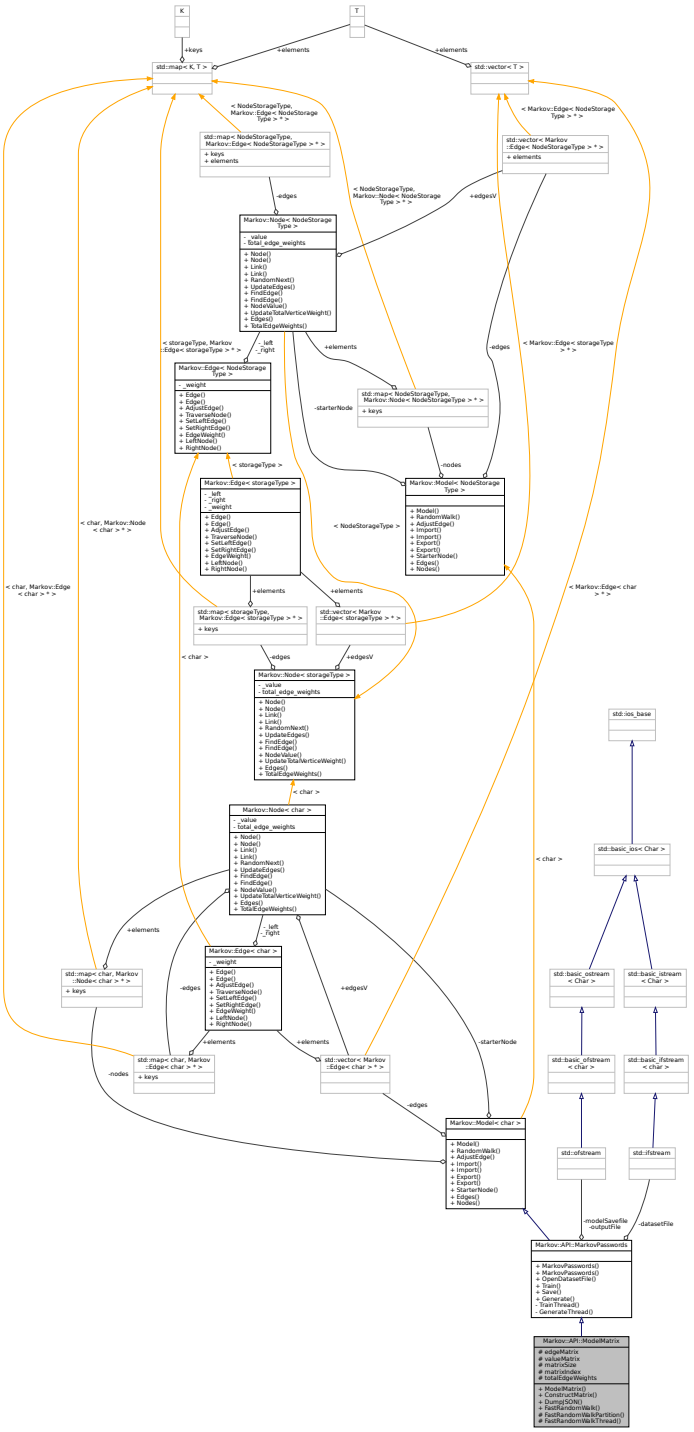
8.16 Markov::API::ModelMatrix Class Reference

Class to flatten and reduce Markov::Model to a Matrix.
`#include <modelMatrix.h>`

Inheritance diagram for Markov::API::ModelMatrix:



Collaboration diagram for Markov::API::ModelMatrix:



Public Member Functions

- ModelMatrix ()
- void ConstructMatrix ()

  Construct the related Matrix data for the model.
- void DumpJSON ()

  Debug function to dump the model to a JSON file.

- void FastRandomWalk (unsigned long int n, const char *wordlistFileName, int minLen=6, int maxLen=12, int threads=20, bool bFileIO=true)

  Random walk on the Matrix-reduced Markov::Model.
- std::ifstream * OpenDatasetFile (const char *filename)

  Open dataset file and return the ifstream pointer.
- void Train (const char *datasetFileName, char delimiter, int threads)

  Train the model with the dataset file.
- std::ofstream * Save (const char *filename)

  Export model to file.
- void Generate (unsigned long int n, const char *wordlistFileName, int minLen=6, int maxLen=12, int threads=20)

  Call Markov::Model::RandomWalk n times, and collect output.
- char * RandomWalk (Markov::Random::RandomEngine *randomEngine, int minSetting, int maxSetting, char *buffer)

  Do a random walk on this model.
- void AdjustEdge (const char *payload, long int occurrence)

  Adjust the model with a single string.
- bool Import (std::ifstream *)

  Import a file to construct the model.
- bool Import (const char *filename)

  Open a file to import with filename, and call bool Model::Import with std::ifstream.
- bool Export (std::ofstream *)

  Export a file of the model.
- bool Export (const char *filename)

  Open a file to export with filename, and call bool Model::Export with std::ofstream.
- Node< char > * StarterNode ()

  Return starter Node.
- std::vector< Edge< char > * > * Edges ()

  Return a vector of all the edges in the model.
- std::map< char, Node< char > * > * Nodes ()

  Return starter Node.

Protected Member Functions

- void FastRandomWalkPartition (std::mutex *mlock, std::ofstream *wordlist, unsigned long int n, int minLen, int maxLen, bool bFileIO, int threads)

  A single partition of FastRandomWalk event.
- void FastRandomWalkThread (std::mutex *mlock, std::ofstream *wordlist, unsigned long int n, int minLen, int maxLen, int id, bool bFileIO)

  A single thread of a single partition of FastRandomWalk.

Protected Attributes

- char ** edgeMatrix

  2-D Character array for the edge Matrix (The characters of Nodes)
- long int ** valueMatrix

  2-d Integer array for the value Matrix (For the weights of Edges)
- int matrixSize

  to hold Matrix size
- char * matrixIndex

  to hold the Matrix index (To hold the orders of 2-D arrays')
- long int * totalEdgeWeights

  Array of the Total Edge Weights.

Private Member Functions

- void TrainThread (Markov::API::Concurrency::ThreadSharedListHandler *listhandler, char delimiter)

  A single thread invoked by the Train function.
- void GenerateThread (std::mutex *outputLock, unsigned long int n, std::ofstream *wordlist, int minLen, int maxLen)

  A single thread invoked by the Generate function.

Private Attributes

- std::ifstream * datasetFile
- std::ofstream * modelSavefile

  Dataset file input of our system
- std::ofstream * outputFile

  File to save model of our system
- std::map< char, Node< char > * > nodes

  Map LeftNode is the Nodes NodeValue Map RightNode is the node pointer.
- Node< char > * starterNode

  Starter Node of this model.
- std::vector< Edge< char > * > edges

  A list of all edges in this model.

8.16.1 Detailed Description

Class to flatten and reduce Markov::Model to a Matrix.
Matrix level operations can be used for Generation events, with a significant performance optimization at the cost of O(N) memory complexity (O(1) memory space for slow mode)
To limit the maximum memory usage, each generation operation is partitioned into 50M chunks for allocation. Threads are sychronized and files are flushed every 50M operations.
Definition at line 13 of file modelMatrix.h.

8.16.2 Constructor & Destructor Documentation

8.16.2.1 ModelMatrix()

```
Markov::API::ModelMatrix::ModelMatrix ( )
```

Definition at line 6 of file modelMatrix.cpp.

```
00006                                   {
00007
00008 }
```

8.16.3 Member Function Documentation

8.16.3.1 AdjustEdge()

```
void Markov::Model< char >::AdjustEdge (
            const NodeStorageType * payload,
            long int occurrence )  [inherited]
```

Adjust the model with a single string.
Start from the starter node, and for each character, AdjustEdge the edge EdgeWeight from current node to the next, until NULL character is reached.
Then, update the edge EdgeWeight from current node, to the terminator node.

This function is used for training purposes, as it can be used for adjusting the model with each line of the corpus file.

Example Use: Create an empty model and train it with string: "testdata"

```
Markov::Model<char> model;
char test[] = "testdata";
model.AdjustEdge(test, 15);
```

Parameters

string	- String that is passed from the training, and will be used to AdjustEdge the model with
occurrence	- Occurrence of this string.

Definition at line 323 of file model.h.

```
00323                                                                                                                {
00324     NodeStorageType p = payload[0];
00325     Markov::Node<NodeStorageType>* curnode = this->starterNode;
00326     Markov::Edge<NodeStorageType>* e;
00327     int i = 0;
00328
00329     if (p == 0) return;
00330     while (p != 0) {
00331         e = curnode->FindEdge(p);
00332         if (e == NULL) return;
00333         e->AdjustEdge(occurrence);
00334         curnode = e->RightNode();
00335         p = payload[++i];
00336     }
00337
00338     e = curnode->FindEdge('\xff');
00339     e->AdjustEdge(occurrence);
00340     return;
00341 }
```

8.16.3.2 ConstructMatrix()

```
void Markov::API::ModelMatrix::ConstructMatrix ( )
```

Construct the related Matrix data for the model.

This operation can be used after importing/training to allocate and populate the matrix content.

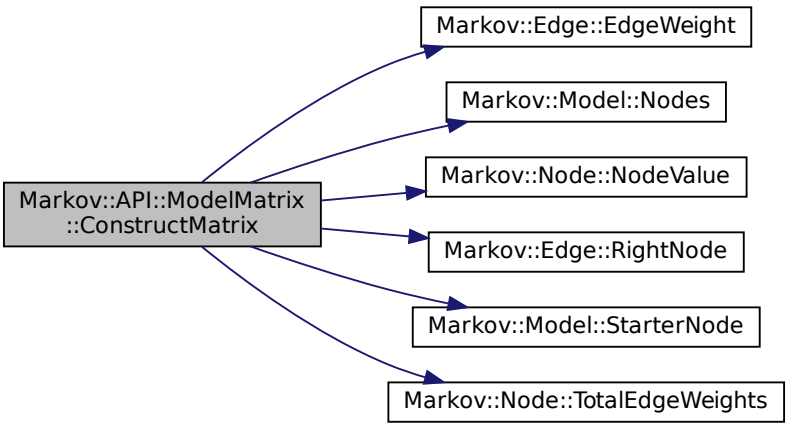
this will initialize: char** edgeMatrix -> a 2D array of mapping left and right connections of each edge. long int **valueMatrix -> a 2D array representing the edge weights. int matrixSize -> Size of the matrix, aka total number of nodes. char* matrixIndex -> order of nodes in the model long int *totalEdgeWeights -> total edge weights of each Node.

Definition at line 11 of file modelMatrix.cpp.

```
00011                                          {
00012     this->matrixSize = this->StarterNode()->edgesV.size() + 2;
00013
00014     this->matrixIndex = new char[this->matrixSize];
00015     this->totalEdgeWeights = new long int[this->matrixSize];
00016
00017     this->edgeMatrix = new char*[this->matrixSize];
00018     for(int i=0;i<this->matrixSize;i++){
00019         this->edgeMatrix[i] = new char[this->matrixSize];
00020     }
00021     this->valueMatrix = new long int*[this->matrixSize];
00022     for(int i=0;i<this->matrixSize;i++){
00023         this->valueMatrix[i] = new long int[this->matrixSize];
00024     }
00025     std::map< char, Node< char > * > *nodes;
00026     nodes = this->Nodes();
00027     int i=0;
00028     for (auto const& [repr, node] : *nodes){
00029         if(repr!=0) this->matrixIndex[i] = repr;
00030         else this->matrixIndex[i] = 199;
00031         this->totalEdgeWeights[i] = node->TotalEdgeWeights();
00032         for(int j=0;j<this->matrixSize;j++){
00033             char val = node->NodeValue();
00034             if(val < 0){
00035                 for(int k=0;k<this->matrixSize;k++){
00036                     this->valueMatrix[i][k] = 0;
00037                     this->edgeMatrix[i][k] = 255;
00038                 }
00039                 break;
00040             }
00041             else if(node->NodeValue() == 0 && j>(this->matrixSize-3)){
00042                 this->valueMatrix[i][j] = 0;
```

```
00043                 this->edgeMatrix[i][j] = 255;
00044             }else if(j==(this->matrixSize-1)) {
00045                 this->valueMatrix[i][j] = 0;
00046                 this->edgeMatrix[i][j] = 255;
00047             }else{
00048                 this->valueMatrix[i][j] = node->edgesV[j]->EdgeWeight();
00049                 this->edgeMatrix[i][j]  = node->edgesV[j]->RightNode()->NodeValue();
00050             }
00051 
00052         }
00053         i++;
00054     }
00055 
00056     //this->DumpJSON();
00057 }
```

References edgeMatrix, Markov::Edge< NodeStorageType >::EdgeWeight(), matrixIndex, matrixSize, Markov::Model< NodeStorage
Markov::Node< storageType >::NodeValue(), Markov::Edge< NodeStorageType >::RightNode(), Markov::Model< NodeStorageTyp
totalEdgeWeights, Markov::Node< storageType >::TotalEdgeWeights(), and valueMatrix.
Referenced by main().
Here is the call graph for this function:



Here is the caller graph for this function:

8.16.3.3 DumpJSON()

```
void Markov::API::ModelMatrix::DumpJSON ( )
```

Debug function to dump the model to a JSON file.
Might not work 100%. Not meant for production use.
Definition at line 60 of file modelMatrix.cpp.

```
00060                                 {
```

```
00061
00062     std::cout « "{\n   \"index\": \"";
00063     for(int i=0;i<this->matrixSize;i++){
00064         if(this->matrixIndex[i]=='"') std::cout « "\\\"";
00065         else if(this->matrixIndex[i]=='\\') std::cout « "\\\\";
00066         else if(this->matrixIndex[i]==0) std::cout « "\\\\x00";
00067         else if(i==0) std::cout « "\\\\xff";
00068         else if(this->matrixIndex[i]=='\n') std::cout « "\\n";
00069         else std::cout « this->matrixIndex[i];
00070     }
00071     std::cout «
00072     "\",\n"
00073     "   \"edgemap\": {\n";
00074
00075     for(int i=0;i<this->matrixSize;i++){
00076         if(this->matrixIndex[i]=='"') std::cout « "      \"\\\"\": [";
00077         else if(this->matrixIndex[i]=='\\') std::cout « "      \"\\\\\": [";
00078         else if(this->matrixIndex[i]==0) std::cout « "      \"\\\\x00\": [";
00079         else if(this->matrixIndex[i]<0) std::cout « "      \"\\\\xff\": [";
00080         else std::cout « "      \"" « this->matrixIndex[i] « "\": [";
00081         for(int j=0;j<this->matrixSize;j++){
00082             if(this->edgeMatrix[i][j]=='"') std::cout « "\"\\\"\"";
00083             else if(this->edgeMatrix[i][j]=='\\') std::cout « "\"\\\\\"";
00084             else if(this->edgeMatrix[i][j]==0) std::cout « "\"\\\\x00\"";
00085             else if(this->edgeMatrix[i][j]<0) std::cout « "\"\\\\xff\"";
00086             else if(this->matrixIndex[i]=='\n') std::cout « "\"\\n\"";
00087             else std::cout « "\"" « this->edgeMatrix[i][j] « "\"";
00088             if(j!=this->matrixSize-1) std::cout « ", ";
00089         }
00090         std::cout « "],\n";
00091     }
00092     std::cout « "},\n";
00093
00094     std::cout « "\"   weightmap\": {\n";
00095     for(int i=0;i<this->matrixSize;i++){
00096         if(this->matrixIndex[i]=='"') std::cout « "      \"\\\"\": [";
00097         else if(this->matrixIndex[i]=='\\') std::cout « "      \"\\\\\": [";
00098         else if(this->matrixIndex[i]==0) std::cout « "      \"\\\\x00\": [";
00099         else if(this->matrixIndex[i]<0) std::cout « "      \"\\\\xff\": [";
00100         else std::cout « "      \"" « this->matrixIndex[i] « "\": [";
00101
00102         for(int j=0;j<this->matrixSize;j++){
00103             std::cout « this->valueMatrix[i][j];
00104             if(j!=this->matrixSize-1) std::cout « ", ";
00105         }
00106         std::cout « "],\n";
00107     }
00108     std::cout « "  }\n}\n";
00109 }
```

References edgeMatrix, matrixIndex, matrixSize, and valueMatrix.

8.16.3.4 Edges()

```
std::vector<Edge<char >*>* Markov::Model< char >::Edges ( )  [inline], [inherited]
```

Return a vector of all the edges in the model.

Returns

vector of edges

Definition at line 172 of file model.h.

```
00172 { return &edges;}
```

8.16.3.5 Export() [1/2]

```
bool Markov::Model< char >::Export (
            const char * filename )  [inherited]
```

Open a file to export with filename, and call bool Model::Export with std::ofstream.

Returns

True if successful, False for incomplete models or corrupt file formats

Example Use: Export file to filename

```
Markov::Model<char> model;
model.Export("test.mdl");
```

Definition at line 286 of file model.h.

```
00286                                                       {
00287     std::ofstream exportfile;
00288     exportfile.open(filename);
00289     return this->Export(&exportfile);
00290 }
```

8.16.3.6 Export() [2/2]

```
bool Markov::Model< char >::Export (
            std::ofstream * f )  [inherited]
```

Export a file of the model.
File contains a list of edges. Format is: Left_repr;EdgeWeight;right_repr. For more information on the format, check out the project wiki or github readme.
Iterate over this vertices, and their edges, and write them to file.

Returns

True if successful, False for incomplete models.

Example Use: Export file to ofstream

```
Markov::Model<char> model;
std::ofstream file("test.mdl");
model.Export(&file);
```

Definition at line 274 of file model.h.

```
00274                                                   {
00275     Markov::Edge<NodeStorageType>* e;
00276     for (std::vector<int>::size_type i = 0; i != this->edges.size(); i++) {
00277         e = this->edges[i];
00278         //std::cout « e->LeftNode()->NodeValue() « "," « e->EdgeWeight() « "," «
      e->RightNode()->NodeValue() « "\n";
00279         *f « e->LeftNode()->NodeValue() « "," « e->EdgeWeight() « "," « e->RightNode()->NodeValue() «
      "\n";
00280     }
00281
00282     return true;
00283 }
```

8.16.3.7 FastRandomWalk()

```
void Markov::API::ModelMatrix::FastRandomWalk (
            unsigned long int n,
            const char * wordlistFileName,
            int minLen = 6,
            int maxLen = 12,
            int threads = 20,
            bool bFileIO = true )
```

Random walk on the Matrix-reduced Markov::Model.
This has an O(N) Memory complexity. To limit the maximum usage, requests with n>50M are partitioned using Markov::API::ModelMatrix::FastRandomWalkPartition.
If n>50M, threads are going to be synced, files are going to be flushed, and buffers will be reallocated every 50M generations. This comes at a minor performance penalty.
While it has the same functionality, this operation reduces Markov::API::MarkovPasswords::Generate runtime by %96.5
This function has deprecated Markov::API::MarkovPasswords::Generate, and will eventually replace it.

Parameters

n	- Number of passwords to generate.
wordlistFileName	- Filename to write to
minLen	- Minimum password length to generate
maxLen	- Maximum password length to generate
threads	- number of OS threads to spawn
bFileIO	- If false, filename will be ignored and will output to stdout.

```
Markov::API::ModelMatrix mp;
mp.Import("models/finished.mdl");
mp.FastRandomWalk(50000000,"./wordlist.txt",6,12,25, true);
```

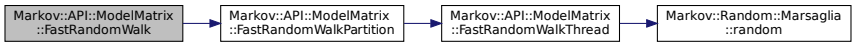
Definition at line 163 of file modelMatrix.cpp.

```
00163
                                          {
00164
00165
00166     std::ofstream wordlist;
00167     if(bFileIO)
00168         wordlist.open(wordlistFileName);
00169
00170     std::mutex mlock;
00171     if(n<=50000000ull) return this->FastRandomWalkPartition(&mlock, &wordlist, n, minLen, maxLen,
      bFileIO, threads);
00172     else{
00173         int numberOfPartitions = n/50000000ull;
00174         for(int i=0;i<numberOfPartitions;i++)
00175             this->FastRandomWalkPartition(&mlock, &wordlist, 50000000ull, minLen, maxLen, bFileIO,
      threads);
00176     }
00177
00178
00179 }
```

References FastRandomWalkPartition().

Referenced by main().

Here is the call graph for this function:



Here is the caller graph for this function:

8.16.3.8 FastRandomWalkPartition()

```
void Markov::API::ModelMatrix::FastRandomWalkPartition (
            std::mutex * mlock,
            std::ofstream * wordlist,
            unsigned long int n,
            int minLen,
            int maxLen,
            bool bFileIO,
            int threads )  [protected]
```

A single partition of FastRandomWalk event.

Since FastRandomWalk has to allocate its output buffer before operation starts and writes data in chunks, large n parameters would lead to huge memory allocations. **Without Partitioning:**

- 50M results 12 characters max -> 550 Mb Memory allocation
- 5B results 12 characters max -> 55 Gb Memory allocation

- 50B results 12 characters max -> 550GB Memory allocation

Instead, FastRandomWalk is partitioned per 50M generations to limit the top memory need.

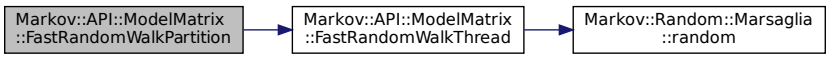
Parameters

mlock	- mutex lock to distribute to child threads
wordlist	- Reference to the wordlist file to write to
n	- Number of passwords to generate.
wordlistFileName	- Filename to write to
minLen	- Minimum password length to generate
maxLen	- Maximum password length to generate
threads	- number of OS threads to spawn
bFileIO	- If false, filename will be ignored and will output to stdout.

Definition at line 182 of file modelMatrix.cpp.

```
00182                                                                    {
00183
00184     int iterationsPerThread = n/threads;
00185     int iterationsPerThreadCarryOver = n%threads;
00186
00187     std::vector<std::thread*> threadsV;
00188
00189     int id = 0;
00190     for(int i=0;i<threads;i++){
00191         threadsV.push_back(new std::thread(&Markov::API::ModelMatrix::FastRandomWalkThread, this,
      mlock, wordlist, iterationsPerThread, minLen, maxLen, id, bFileIO));
00192         id++;
00193     }
00194
00195     threadsV.push_back(new std::thread(&Markov::API::ModelMatrix::FastRandomWalkThread, this, mlock,
      wordlist, iterationsPerThreadCarryOver, minLen, maxLen, id, bFileIO));
00196
00197     for(int i=0;i<threads;i++){
00198         threadsV[i]->join();
00199     }
00200 }
```

References FastRandomWalkThread().
Referenced by FastRandomWalk().
Here is the call graph for this function:



Here is the caller graph for this function:

8.16.3.9 FastRandomWalkThread()

```
void Markov::API::ModelMatrix::FastRandomWalkThread (
            std::mutex * mlock,
```

```
            std::ofstream * wordlist,
            unsigned long int n,
            int minLen,
            int maxLen,
            int id,
            bool bFileIO )  [protected]
```

A single thread of a single partition of FastRandomWalk.
A FastRandomWalkPartition will initiate as many of this function as requested.
This function contains the bulk of the generation algorithm.

Parameters

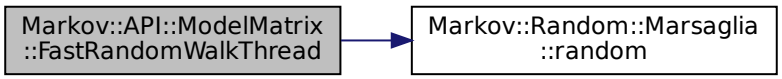
mlock	- mutex lock to distribute to child threads
wordlist	- Reference to the wordlist file to write to
n	- Number of passwords to generate.
wordlistFileName	- Filename to write to
minLen	- Minimum password length to generate
maxLen	- Maximum password length to generate
id	- **DEPRECATED** Thread id - No longer used
bFileIO	- If false, filename will be ignored and will output to stdout.

Definition at line 112 of file modelMatrix.cpp.

```
00112                                                       {
00113     if(n==0) return;
00114
00115     Markov::Random::Marsaglia MarsagliaRandomEngine;
00116     char* e;
00117     char *res = new char[maxLen*n];
00118     int index = 0;
00119     char next;
00120     int len=0;
00121     long int selection;
00122     char cur;
00123     long int bufferctr = 0;
00124     for (int i = 0; i < n; i++) {
00125         cur=199;
00126         len=0;
00127         while (true) {
00128             e = strchr(this->matrixIndex, cur);
00129             index = e - this->matrixIndex;
00130             selection = MarsagliaRandomEngine.random() % this->totalEdgeWeights[index];
00131             for(int j=0;j<this->matrixSize;j++){
00132                 selection -= this->valueMatrix[index][j];
00133                 if (selection < 0){
00134                     next = this->edgeMatrix[index][j];
00135                     break;
00136                 }
00137             }
00138
00139             if (len >= maxLen)  break;
00140             else if ((next < 0) && (len < minLen)) continue;
00141             else if (next < 0) break;
00142             cur = next;
00143             res[bufferctr + len++] = cur;
00144         }
00145         res[bufferctr + len++] = '\n';
00146         bufferctr+=len;
00147
00148     }
00149     if(bFileIO){
00150         mlock->lock();
00151         *wordlist << res;
00152         mlock->unlock();
00153     }else{
00154         mlock->lock();
00155         std::cout << res;
00156         mlock->unlock();
00157     }
00158     delete res;
00159
00160 }
```

References edgeMatrix, matrixIndex, matrixSize, Markov::Random::Marsaglia::random(), totalEdgeWeights, and valueMatrix.

Referenced by FastRandomWalkPartition().
Here is the call graph for this function:



Here is the caller graph for this function:

8.16.3.10 Generate()

```
void Markov::API::MarkovPasswords::Generate (
            unsigned long int n,
            const char * wordlistFileName,
            int minLen = 6,
            int maxLen = 12,
            int threads = 20 )  [inherited]
```

Call Markov::Model::RandomWalk n times, and collect output.
Generate from model and write results to a file. a much more performance-optimized method. FastRandomWalk will reduce the runtime by %96.5 on average.

Deprecated See Markov::API::MatrixModel::FastRandomWalk for more information.

Parameters

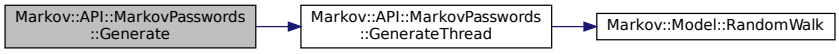
n	- Number of passwords to generate.
wordlistFileName	- Filename to write to
minLen	- Minimum password length to generate
maxLen	- Maximum password length to generate
threads	- number of OS threads to spawn

Definition at line 110 of file markovPasswords.cpp.

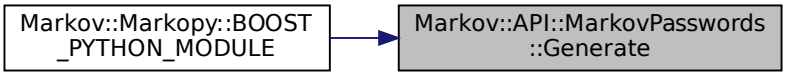
```
00110                                {
00111     char* res;
00112     char print[100];
00113     std::ofstream wordlist;
00114     wordlist.open(wordlistFileName);
00115     std::mutex mlock;
00116     int iterationsPerThread = n/threads;
00117     int iterationsCarryOver = n%threads;
00118     std::vector<std::thread*> threadsV;
00119     for(int i=0;i<threads;i++){
00120         threadsV.push_back(new std::thread(&Markov::API::MarkovPasswords::GenerateThread, this,
       &mlock, iterationsPerThread, &wordlist, minLen, maxLen));
00121     }
00122
00123     for(int i=0;i<threads;i++){
00124         threadsV[i]->join();
```

```
00125         delete threadsV[i];
00126     }
00127
00128     this->GenerateThread(&mlock, iterationsCarryOver, &wordlist, minLen, maxLen);
00129
00130 }
```

References Markov::API::MarkovPasswords::GenerateThread().
Referenced by Markov::Markopy::BOOST_PYTHON_MODULE().
Here is the call graph for this function:



Here is the caller graph for this function:



8.16.3.11 GenerateThread()

```
void Markov::API::MarkovPasswords::GenerateThread (
              std::mutex * outputLock,
              unsigned long int n,
              std::ofstream * wordlist,
              int minLen,
              int maxLen )  [private], [inherited]
```

A single thread invoked by the Generate function.
DEPRECATED: See Markov::API::MatrixModel::FastRandomWalkThread for more information. This has been replaced with a much more performance-optimized method. FastRandomWalk will reduce the runtime by %96.5 on average.

Parameters

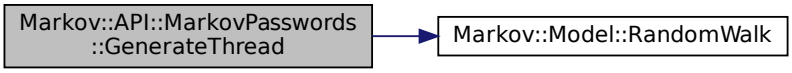
outputLock	- shared mutex lock to lock during output operation. Prevents race condition on write.
n	number of lines to be generated by this thread
wordlist	wordlistfile
minLen	- Minimum password length to generate
maxLen	- Maximum password length to generate

Definition at line 132 of file markovPasswords.cpp.

```
00132                                       {
00133     char* res = new char[maxLen+5];
00134     if(n==0) return;
00135
00136     Markov::Random::Marsaglia MarsagliaRandomEngine;
00137     for (int i = 0; i < n; i++) {
00138         this->RandomWalk(&MarsagliaRandomEngine, minLen, maxLen, res);
00139         outputLock->lock();
00140         *wordlist « res « "\n";
00141         outputLock->unlock();
00142     }
```

```
00143 }
```

References Markov::Model< NodeStorageType >::RandomWalk().
Referenced by Markov::API::MarkovPasswords::Generate().
Here is the call graph for this function:



Here is the caller graph for this function:

8.16.3.12 Import() [1/2]

```
bool Markov::Model< char >::Import (
            const char * filename )  [inherited]
```

Open a file to import with filename, and call bool Model::Import with std::ifstream.

Returns

True if successful, False for incomplete models or corrupt file formats

Example Use: Import a file with filename

```
Markov::Model<char> model;
model.Import("test.mdl");
```

Definition at line 266 of file model.h.

```
00266                                                    {
00267     std::ifstream importfile;
00268     importfile.open(filename);
00269     return this->Import(&importfile);
00270
00271 }
```

8.16.3.13 Import() [2/2]

```
bool Markov::Model< char >::Import (
            std::ifstream * f )  [inherited]
```

Import a file to construct the model.
File contains a list of edges. For more info on the file format, check out the wiki and github readme pages. Format is: Left_repr;EdgeWeight;right_repr
Iterate over this list, and construct nodes and edges accordingly.

Returns

True if successful, False for incomplete models or corrupt file formats

Example Use: Import a file from ifstream

```
Markov::Model<char> model;
std::ifstream file("test.mdl");
model.Import(&file);
```

Definition at line 207 of file model.h.

```
00207                                                {
```

```
00208     std::string cell;
00209
00210     char src;
00211     char target;
00212     long int oc;
00213
00214     while (std::getline(*f, cell)) {
00215         //std::cout « "cell: " « cell « std::endl;
00216         src = cell[0];
00217         target = cell[cell.length() - 1];
00218         char* j;
00219         oc = std::strtol(cell.substr(2, cell.length() - 2).c_str(),&j,10);
00220         //std::cout « oc « "\n";
00221         Markov::Node<NodeStorageType>* srcN;
00222         Markov::Node<NodeStorageType>* targetN;
00223         Markov::Edge<NodeStorageType>* e;
00224         if (this->nodes.find(src) == this->nodes.end()) {
00225             srcN = new Markov::Node<NodeStorageType>(src);
00226             this->nodes.insert(std::pair<char, Markov::Node<NodeStorageType>*>(src, srcN));
00227             //std::cout « "Creating new node at start.\n";
00228         }
00229         else {
00230             srcN = this->nodes.find(src)->second;
00231         }
00232
00233         if (this->nodes.find(target) == this->nodes.end()) {
00234             targetN = new Markov::Node<NodeStorageType>(target);
00235             this->nodes.insert(std::pair<char, Markov::Node<NodeStorageType>*>(target, targetN));
00236             //std::cout « "Creating new node at end.\n";
00237         }
00238         else {
00239             targetN = this->nodes.find(target)->second;
00240         }
00241         e = srcN->Link(targetN);
00242         e->AdjustEdge(oc);
00243         this->edges.push_back(e);
00244
00245         //std::cout « int(srcN->NodeValue()) « " --" « e->EdgeWeight() « "--> " «
      int(targetN->NodeValue()) « "\n";
00246
00247
00248     }
00249
00250     for (std::pair<unsigned char, Markov::Node<NodeStorageType>*> const& x : this->nodes) {
00251         //std::cout « "Total edges in EdgesV: " « x.second->edgesV.size() « "\n";
00252         std::sort (x.second->edgesV.begin(), x.second->edgesV.end(), [](Edge<NodeStorageType> *lhs,
      Edge<NodeStorageType> *rhs)->bool{
00253             return lhs->EdgeWeight() > rhs->EdgeWeight();
00254         });
00255         //for(int i=0;i<x.second->edgesV.size();i++)
00256         //  std::cout « x.second->edgesV[i]->EdgeWeight() « ", ";
00257         //std::cout « "\n";
00258     }
00259     //std::cout « "Total number of nodes: " « this->nodes.size() « std::endl;
00260     //std::cout « "Total number of edges: " « this->edges.size() « std::endl;
00261
00262     return true;
00263 }
```

8.16.3.14 Nodes()

`std::map<char , Node<char >*>* Markov::Model< char >::Nodes ( )  [inline], [inherited]`

Return starter Node.

Returns

starter node with 00 NodeValue

Definition at line 177 of file model.h.

```
00177 { return &nodes;}
```

8.16.3.15 OpenDatasetFile()

```
std::ifstream * Markov::API::MarkovPasswords::OpenDatasetFile (
            const char * filename )  [inherited]
```

Open dataset file and return the ifstream pointer.

Parameters

filename	- Filename to open

Returns

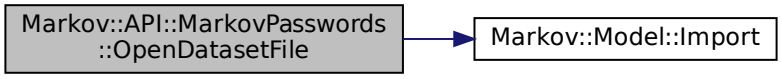
ifstream* to the the dataset file

Definition at line 43 of file markovPasswords.cpp.

```
00043                                                                    {
00044
00045     std::ifstream* datasetFile;
00046
00047     std::ifstream newFile(filename);
00048
00049     datasetFile = &newFile;
00050
00051     this->Import(datasetFile);
00052     return datasetFile;
00053 }
```

References Markov::Model< NodeStorageType >::Import().

Here is the call graph for this function:



8.16.3.16 RandomWalk()

```
char * Markov::Model< char >::RandomWalk (
            Markov::Random::RandomEngine * randomEngine,
            int minSetting,
            int maxSetting,
            NodeStorageType * buffer )  [inherited]
```

Do a random walk on this model.

Start from the starter node, on each node, invoke RandomNext using the random engine on current node, until terminator node is reached. If terminator node is reached before minimum length criateria is reached, ignore the last selection and re-invoke randomNext

If maximum length criteria is reached but final node is not, cut off the generation and proceed to the final node. This function takes Markov::Random::RandomEngine as a parameter to generate pseudo random numbers from

This library is shipped with two random engines, Marsaglia and Mersenne. While mersenne output is higher in entropy, most use cases don't really need super high entropy output, so Markov::Random::Marsaglia is preferable for better performance.

This function WILL NOT reallocate buffer. Make sure no out of bound writes are happening via maximum length criteria.

Example Use: Generate 10 lines, with 5 to 10 characters, and print the output. Use Marsaglia

```
Markov::Model<char> model;
Model.import("model.mdl");
char* res = new char[11];
Markov::Random::Marsaglia MarsagliaRandomEngine;
for (int i = 0; i < 10; i++) {
    this->RandomWalk(&MarsagliaRandomEngine, 5, 10, res);
    std::cout « res « "\n";
 }
```

Parameters

randomEngine	Random Engine to use for the random walks. For examples, see Markov::Random::Mersenne and Markov::Random::Marsaglia
minSetting	Minimum number of characters to generate
maxSetting	Maximum number of character to generate
buffer	buffer to write the result to

Returns

Null terminated string that was generated.

Definition at line 293 of file model.h.

```
00293                                                                {
00294     Markov::Node<NodeStorageType>* n = this->starterNode;
00295     int len = 0;
00296     Markov::Node<NodeStorageType>* temp_node;
00297     while (true) {
00298         temp_node = n->RandomNext(randomEngine);
00299         if (len >= maxSetting) {
00300             break;
00301         }
00302         else if ((temp_node == NULL) && (len < minSetting)) {
00303             continue;
00304         }
00305
00306         else if (temp_node == NULL){
00307             break;
00308         }
00309
00310         n = temp_node;
00311
00312         buffer[len++] = n->NodeValue();
00313     }
00314
00315     //null terminate the string
00316     buffer[len] = 0x00;
00317
00318     //do something with the generated string
00319     return buffer; //for now
00320 }
```

8.16.3.17 Save()

```
std::ofstream * Markov::API::MarkovPasswords::Save (
            const char * filename )  [inherited]
```

Export model to file.

Parameters

filename	- Export filename.

Returns

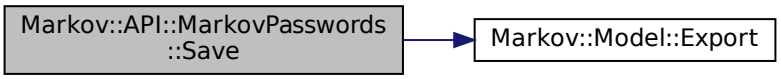
std::ofstream* of the exported file.

Definition at line 98 of file markovPasswords.cpp.

```
00098                                                                {
00099     std::ofstream* exportFile;
00100
00101     std::ofstream newFile(filename);
00102
00103     exportFile = &newFile;
00104
00105     this->Export(exportFile);
00106     return exportFile;
00107 }
```

References Markov::Model< NodeStorageType >::Export().

Here is the call graph for this function:



8.16.3.18 StarterNode()

```
Node<char >* Markov::Model< char >::StarterNode ( )  [inline], [inherited]
```

Return starter Node.

Returns

starter node with 00 NodeValue

Definition at line 167 of file model.h.

```
00167 { return starterNode;}
```

8.16.3.19 Train()

```
void Markov::API::MarkovPasswords::Train (
            const char * datasetFileName,
            char delimiter,
            int threads )  [inherited]
```

Train the model with the dataset file.

Parameters

datasetFileName	- Ifstream* to the dataset. If null, use class member
delimiter	- a character, same as the delimiter in dataset content
threads	- number of OS threads to spawn

```
Markov::API::MarkovPasswords mp;
mp.Import("models/2gram.mdl");
mp.Train("password.corpus");
```

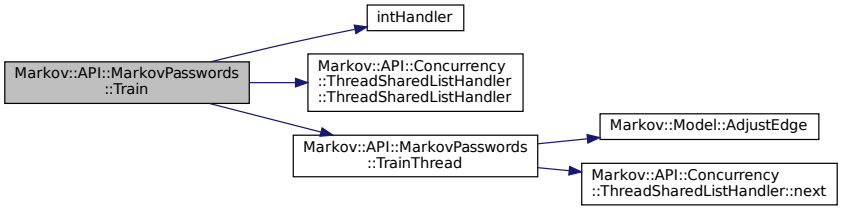
Definition at line 57 of file markovPasswords.cpp.

```
00057                                                                                                {
00058   signal(SIGINT, intHandler);
00059     Markov::API::Concurrency::ThreadSharedListHandler listhandler(datasetFileName);
00060     auto start = std::chrono::high_resolution_clock::now();
00061
00062     std::vector<std::thread*> threadsV;
00063     for(int i=0;i<threads;i++){
00064         threadsV.push_back(new std::thread(&Markov::API::MarkovPasswords::TrainThread, this,
      &listhandler, delimiter));
00065     }
00066
00067     for(int i=0;i<threads;i++){
00068         threadsV[i]->join();
00069         delete threadsV[i];
00070     }
00071     auto finish = std::chrono::high_resolution_clock::now();
00072     std::chrono::duration<double> elapsed = finish - start;
00073     std::cout << "Elapsed time: " << elapsed.count() << " s\n";
00074
00075 }
```

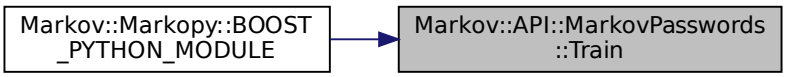
References intHandler(), Markov::API::Concurrency::ThreadSharedListHandler::ThreadSharedListHandler(), and Markov::API::MarkovPasswords::TrainThread().

Referenced by Markov::Markopy::BOOST_PYTHON_MODULE().

Here is the call graph for this function:



Here is the caller graph for this function:



8.16.3.20 TrainThread()

```
void Markov::API::MarkovPasswords::TrainThread (
            Markov::API::Concurrency::ThreadSharedListHandler * listhandler,
            char delimiter )  [private], [inherited]
```

A single thread invoked by the Train function.

Parameters

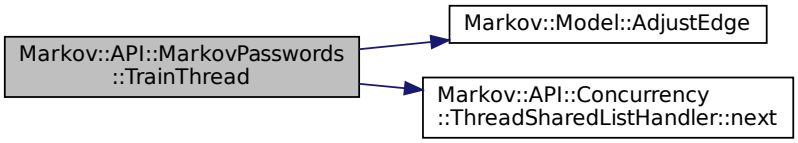
listhandler	- Listhandler class to read corpus from
delimiter	- a character, same as the delimiter in dataset content

Definition at line 77 of file markovPasswords.cpp.

```
00077                {
00078     char format_str[] ="%ld,%s";
00079     format_str[2]=delimiter;
00080     std::string line;
00081     while (listhandler->next(&line) && keepRunning) {
00082         long int oc;
00083         if (line.size() > 100) {
00084             line = line.substr(0, 100);
00085         }
00086         char* linebuf = new char[line.length()+5];
00087 #ifdef _WIN32
00088         sscanf_s(line.c_str(), "%ld,%s", &oc, linebuf, line.length()+5); //<== changed format_str to->
       "%ld,%s"
00089 #else
00090         sscanf(line.c_str(), format_str, &oc, linebuf);
00091 #endif
00092         this->AdjustEdge((const char*)linebuf, oc);
00093         delete linebuf;
00094     }
00095 }
```

References Markov::Model< NodeStorageType >::AdjustEdge(), keepRunning, and Markov::API::Concurrency::ThreadSharedListHa
Referenced by Markov::API::MarkovPasswords::Train().

Here is the call graph for this function:



Here is the caller graph for this function:

8.16.4 Member Data Documentation

8.16.4.1 datasetFile

`std::ifstream* Markov::API::MarkovPasswords::datasetFile [private], [inherited]`
Definition at line 106 of file markovPasswords.h.

8.16.4.2 edgeMatrix

`char** Markov::API::ModelMatrix::edgeMatrix [protected]`
2-D Character array for the edge Matrix (The characters of Nodes)
Definition at line 116 of file modelMatrix.h.
Referenced by ConstructMatrix(), DumpJSON(), and FastRandomWalkThread().

8.16.4.3 edges

`std::vector<Edge<char >*> Markov::Model< char >::edges [private], [inherited]`
A list of all edges in this model.
Definition at line 195 of file model.h.

8.16.4.4 matrixIndex

`char* Markov::API::ModelMatrix::matrixIndex [protected]`
to hold the Matrix index (To hold the orders of 2-D arrays')
Definition at line 131 of file modelMatrix.h.
Referenced by ConstructMatrix(), DumpJSON(), and FastRandomWalkThread().

8.16.4.5 matrixSize

`int Markov::API::ModelMatrix::matrixSize [protected]`
to hold Matrix size
Definition at line 126 of file modelMatrix.h.

Referenced by ConstructMatrix(), DumpJSON(), and FastRandomWalkThread().

8.16.4.6 modelSavefile

`std::ofstream* Markov::API::MarkovPasswords::modelSavefile [private], [inherited]`
Dataset file input of our system

Definition at line 107 of file markovPasswords.h.

8.16.4.7 nodes

`std::map<char , Node<char >*> Markov::Model< char >::nodes [private], [inherited]`
Map LeftNode is the Nodes NodeValue Map RightNode is the node pointer.
Definition at line 184 of file model.h.

8.16.4.8 outputFile

`std::ofstream* Markov::API::MarkovPasswords::outputFile [private], [inherited]`
File to save model of our system

Definition at line 108 of file markovPasswords.h.

8.16.4.9 starterNode

`Node<char >* Markov::Model< char >::starterNode [private], [inherited]`
Starter Node of this model.
Definition at line 189 of file model.h.

8.16.4.10 totalEdgeWeights

`long int* Markov::API::ModelMatrix::totalEdgeWeights [protected]`
Array of the Total Edge Weights.
Definition at line 136 of file modelMatrix.h.
Referenced by ConstructMatrix(), and FastRandomWalkThread().

8.16.4.11 valueMatrix

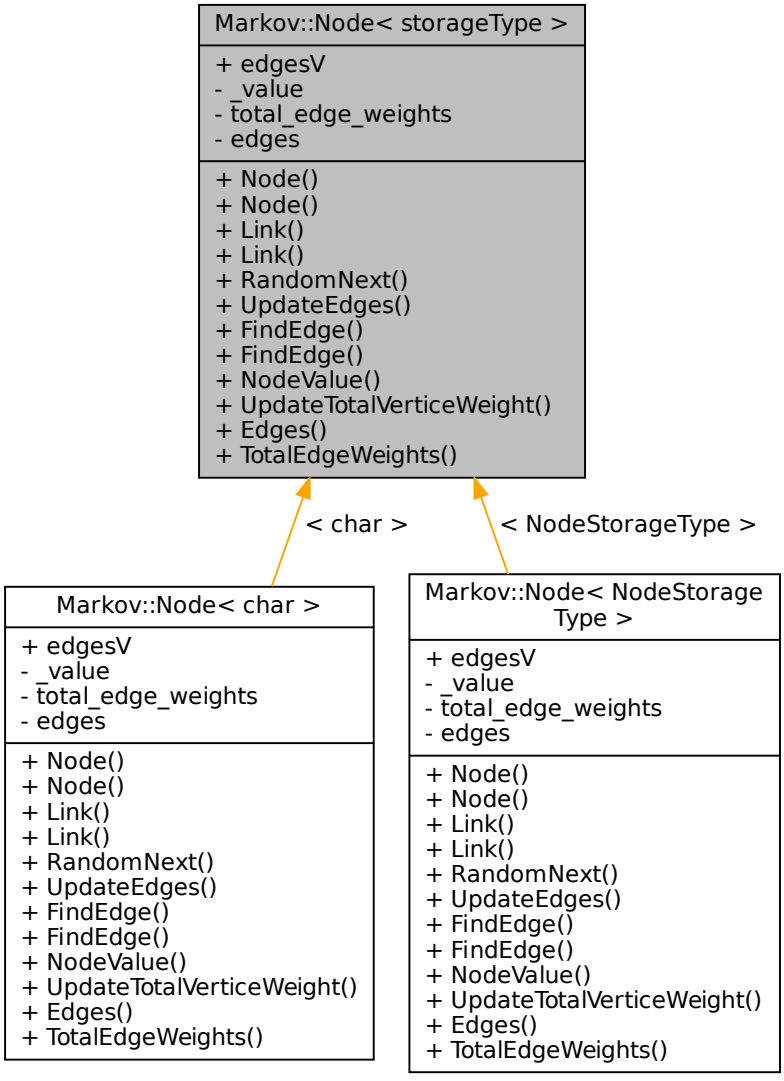
`long int** Markov::API::ModelMatrix::valueMatrix [protected]`
2-d Integer array for the value Matrix (For the weights of Edges)
Definition at line 121 of file modelMatrix.h.
Referenced by ConstructMatrix(), DumpJSON(), and FastRandomWalkThread().
The documentation for this class was generated from the following files:

- modelMatrix.h
- modelMatrix.cpp

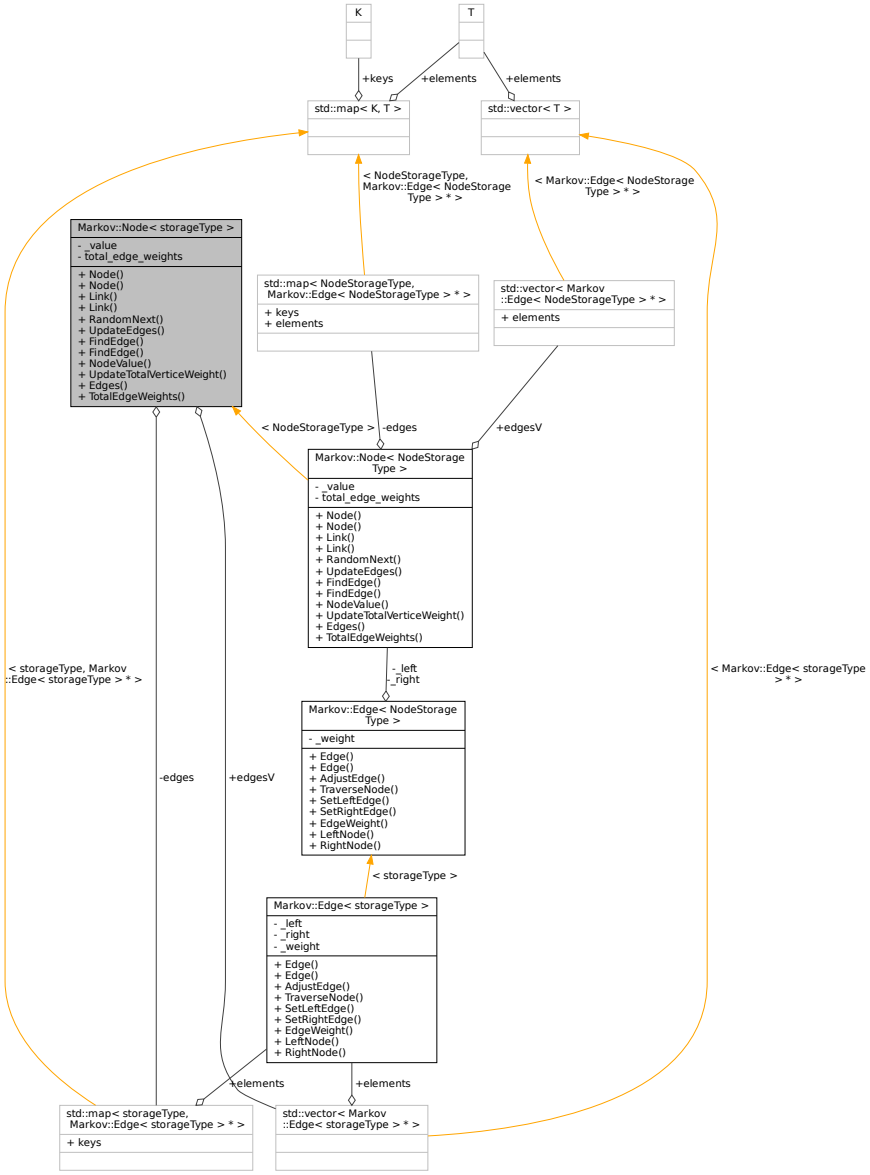
8.17 Markov::Node< storageType > Class Template Reference

A node class that for the vertices of model. Connected with eachother using Edge.
`#include <model.h>`

Inheritance diagram for Markov::Node< storageType >:



Collaboration diagram for Markov::Node< storageType >:



Public Member Functions

- Node ()

  Default constructor. Creates an empty Node.
- Node (storageType _value)

  Constructor. Creates a Node with no edges and with given NodeValue.
- Edge< storageType > * Link (Node< storageType > *)

  Link this node with another, with this node as its source.
- Edge< storageType > * Link (Edge< storageType > *)

  Link this node with another, with this node as its source.
- Node< storageType > * RandomNext (Markov::Random::RandomEngine *randomEngine)

  Chose a random node from the list of edges, with regards to its EdgeWeight, and TraverseNode to that.
- bool UpdateEdges (Edge< storageType > *)

  Insert a new edge to the this.edges.

- Edge< storageType > * FindEdge (storageType repr)

  Find an edge with its character representation.
- Edge< storageType > * FindEdge (Node< storageType > *target)

  Find an edge with its pointer. Avoid unless neccessary because comptutational cost of find by character is cheaper (because of std::map)
- unsigned char NodeValue ()

  Return character representation of this node.
- void UpdateTotalVerticeWeight (long int offset)

  Change total weights with offset.
- std::map< storageType, Edge< storageType > * > * Edges ()

  return edges
- long int TotalEdgeWeights ()

  return total edge weights

Public Attributes

- std::vector< Edge< storageType > * > edgesV

Private Attributes

- storageType _value

  Character representation of this node. 0 for starter, 0xff for terminator.
- long int total_edge_weights

  Total weights of the vertices, required by RandomNext.
- std::map< storageType, Edge< storageType > * > edges

  A map of all edges connected to this node, where this node is at the LeftNode. Map is indexed by unsigned char, which is the character representation of the node.

8.17.1 Detailed Description

template<typename storageType>
class Markov::Node< storageType >

A node class that for the vertices of model. Connected with eachother using Edge.
This class will later be templated to accept other data types than char*.
Definition at line 23 of file model.h.

8.17.2 Constructor & Destructor Documentation

8.17.2.1 Node() [1/2]

```
template<typename storageType >
Markov::Node< storageType >::Node
```

Default constructor. Creates an empty Node.
Definition at line 201 of file node.h.

```
00201                                {
00202     this->_value = 0;
00203     this->total_edge_weights = 0L;
00204 };
```

8.17.2.2 Node() [2/2]

```
template<typename storageType >
Markov::Node< storageType >::Node (
            storageType _value )
```

Constructor. Creates a Node with no edges and with given NodeValue.

Parameters

_value	- Nodes character representation.

Example Use: Construct nodes

```
Markov::Node<unsigned char>* LeftNode = new Markov::Node<unsigned char>('l');
Markov::Node<unsigned char>* RightNode = new Markov::Node<unsigned char>('r');
```

Definition at line 195 of file node.h.

```
00195                                                {
00196     this->_value = _value;
00197     this->total_edge_weights = 0L;
00198 };
```

8.17.3 Member Function Documentation

8.17.3.1 Edges()

```
template<typename storageType >
std::map< storageType, Markov::Edge< storageType > * > * Markov::Node< storageType >::Edges
[inline]
```

return edges

Definition at line 264 of file node.h.

```
00264                                                                                {
00265     return &(this->edges);
00266 }
```

8.17.3.2 FindEdge() [1/2]

```
template<typename storageType >
Edge<storageType>* Markov::Node< storageType >::FindEdge (
            Node< storageType > * target )
```

Find an edge with its pointer. Avoid unless neccessary because comptutational cost of find by character is cheaper (because of std::map)

Parameters

target	- target node.

Returns

Edge that is connected between this node, and the target node.

8.17.3.3 FindEdge() [2/2]

```
template<typename storageType >
Markov::Edge< storageType > * Markov::Node< storageType >::FindEdge (
            storageType repr )
```

Find an edge with its character representation.

Parameters

repr	- character NodeValue of the target node.

Returns

Edge that is connected between this node, and the target node.

Example Use: Construct and update edges

```
Markov::Node<unsigned char>* src = new Markov::Node<unsigned char>('a');
Markov::Node<unsigned char>* target1 = new Markov::Node<unsigned char>('b');
Markov::Node<unsigned char>* target2 = new Markov::Node<unsigned char>('c');
Markov::Edge<unsigned char>* res = NULL;
src->Link(target1);
src->Link(target2);
res = src->FindEdge('b');
```

Definition at line 252 of file node.h.

```
00252                                                                    {
00253     auto e = this->edges.find(repr);
00254     if (e == this->edges.end()) return NULL;
00255     return e->second;
00256 };
```

8.17.3.4 Link() [1/2]

```
template<typename storageType >
Markov::Edge< storageType > * Markov::Node< storageType >::Link (
            Markov::Edge< storageType > * v )
```

Link this node with another, with this node as its source.
DOES NOT create a new Edge.

Parameters

Edge	- Edge that will accept this node as its LeftNode.

Returns

the same edge as parameter target.

Example Use: Construct and link nodes

```
Markov::Node<unsigned char>* LeftNode = new Markov::Node<unsigned char>('l');
Markov::Node<unsigned char>* RightNode = new Markov::Node<unsigned char>('r');
Markov::Edge<unsigned char>* e = LeftNode->Link(RightNode);
LeftNode->Link(e);
```

Definition at line 219 of file node.h.

```
00219                                                                    {
00220     v->SetLeftEdge(this);
00221     this->UpdateEdges(v);
00222     return v;
00223 }
```

8.17.3.5 Link() [2/2]

```
template<typename storageType >
Markov::Edge< storageType > * Markov::Node< storageType >::Link (
            Markov::Node< storageType > * n )
```

Link this node with another, with this node as its source.
Creates a new Edge.

Parameters

target	- Target node which will be the RightNode() of new edge.

Returns

A new node with LeftNode as this, and RightNode as parameter target.

Example Use: Construct nodes

```
Markov::Node<unsigned char>* LeftNode = new Markov::Node<unsigned char>('l');
```

```
Markov::Node<unsigned char>* RightNode = new Markov::Node<unsigned char>('r');
Markov::Edge<unsigned char>* e = LeftNode->Link(RightNode);
```

Definition at line 212 of file node.h.

```
00212                                                                    {
00213     Markov::Edge<storageType>* v = new Markov::Edge<storageType>(this, n);
00214     this->UpdateEdges(v);
00215     return v;
00216 }
```

8.17.3.6 NodeValue()

```
template<typename storageType >
unsigned char Markov::Node< storageType >::NodeValue  [inline]
```

Return character representation of this node.

Returns

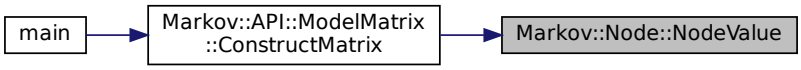
character representation at _value.

Definition at line 207 of file node.h.

```
00207                                                {
00208     return _value;
00209 }
```

Referenced by Markov::API::ModelMatrix::ConstructMatrix().

Here is the caller graph for this function:



8.17.3.7 RandomNext()

```
template<typename storageType >
Markov::Node< storageType > * Markov::Node< storageType >::RandomNext (
            Markov::Random::RandomEngine * randomEngine )
```

Chose a random node from the list of edges, with regards to its EdgeWeight, and TraverseNode to that.
This operation is done by generating a random number in range of 0-this.total_edge_weights, and then iterating over the list of edges. At each step, EdgeWeight of the edge is subtracted from the random number, and once it is 0, next node is selected.

Returns

Node that was chosen at EdgeWeight biased random.

Example Use: Use randomNext to do a random walk on the model

```
char* buffer[64];
Markov::Model<char> model;
model.Import("model.mdl");
   Markov::Node<char>* n = model.starterNode;
int len = 0;
Markov::Node<char>* temp_node;
while (true) {
   temp_node = n->RandomNext(randomEngine);
   if (len >= maxSetting) {
       break;
   }
   else if ((temp_node == NULL) && (len < minSetting)) {
       continue;
   }
   else if (temp_node == NULL){
       break;
   }
```

```
   n = temp_node;
   buffer[len++] = n->NodeValue();
}
```

Definition at line 226 of file node.h.

```
00226                                                                                                {
00227
00228     //get a random NodeValue in range of total_vertice_weight
00229     long int selection = randomEngine->random() %
      this->total_edge_weights;//distribution()(generator());// distribution(generator);
00230     //make absolute, no negative modulus values wanted
00231     //selection = (selection >= 0) ? selection : (selection + this->total_edge_weights);
00232     for(int i=0;i<this->edgesV.size();i++){
00233         selection -= this->edgesV[i]->EdgeWeight();
00234         if (selection < 0) return this->edgesV[i]->TraverseNode();
00235     }
00236
00237     //if this assertion is reached, it means there is an implementation error above
00238     std::cout << "This should never be reached (node failed to walk to next)\n"; //cant assert from
      child thread
00239     assert(true && "This should never be reached (node failed to walk to next)");
00240     return NULL;
00241 }
```

8.17.3.8 TotalEdgeWeights()

```
template<typename storageType >
long int Markov::Node< storageType >::TotalEdgeWeights  [inline]
```

return total edge weights

Definition at line 269 of file node.h.

```
00269                                                  {
00270     return this->total_edge_weights;
00271 }
```

Referenced by Markov::API::ModelMatrix::ConstructMatrix().

Here is the caller graph for this function:

8.17.3.9 UpdateEdges()

```
template<typename storageType >
bool Markov::Node< storageType >::UpdateEdges (
            Markov::Edge< storageType > * v )
```

Insert a new edge to the this.edges.

Parameters

edge	- New edge that will be inserted.

Returns

true if insertion was successful, false if it fails.

Example Use: Construct and update edges

```
Markov::Node<unsigned char>* src = new Markov::Node<unsigned char>('a');
Markov::Node<unsigned char>* target1 = new Markov::Node<unsigned char>('b');
Markov::Node<unsigned char>* target2 = new Markov::Node<unsigned char>('c');
Markov::Edge<unsigned char>* e1 = new Markov::Edge<unsigned char>(src, target1);
Markov::Edge<unsigned char>* e2 = new Markov::Edge<unsigned char>(src, target2);
e1->AdjustEdge(25);
src->UpdateEdges(e1);
```

```
e2->AdjustEdge(30);
src->UpdateEdges(e2);
```

Definition at line 244 of file node.h.

```
00244                                                                    {
00245     this->edges.insert({ v->RightNode()->NodeValue(), v });
00246     this->edgesV.push_back(v);
00247     //this->total_edge_weights += v->EdgeWeight();
00248     return v->TraverseNode();
00249 }
```

8.17.3.10 UpdateTotalVerticeWeight()

```
template<typename storageType >
void Markov::Node< storageType >::UpdateTotalVerticeWeight (
            long int offset )
```

Change total weights with offset.

Parameters

offset	to adjust the vertice weight with

Definition at line 259 of file node.h.

```
00259                                                                    {
00260     this->total_edge_weights += offset;
00261 }
```

8.17.4 Member Data Documentation

8.17.4.1 _value

```
template<typename storageType >
storageType Markov::Node< storageType >::_value  [private]
```

Character representation of this node. 0 for starter, 0xff for terminator.

Definition at line 171 of file node.h.

Referenced by Markov::Node< NodeStorageType >::NodeValue().

8.17.4.2 edges

```
template<typename storageType >
std::map<storageType, Edge<storageType>*> Markov::Node< storageType >::edges  [private]
```

A map of all edges connected to this node, where this node is at the LeftNode. Map is indexed by unsigned char, which is the character representation of the node.

Definition at line 182 of file node.h.

8.17.4.3 edgesV

```
template<typename storageType >
std::vector<Edge<storageType>*> Markov::Node< storageType >::edgesV
```

Definition at line 165 of file node.h.

8.17.4.4 total_edge_weights

```
template<typename storageType >
long int Markov::Node< storageType >::total_edge_weights  [private]
```

Total weights of the vertices, required by RandomNext.

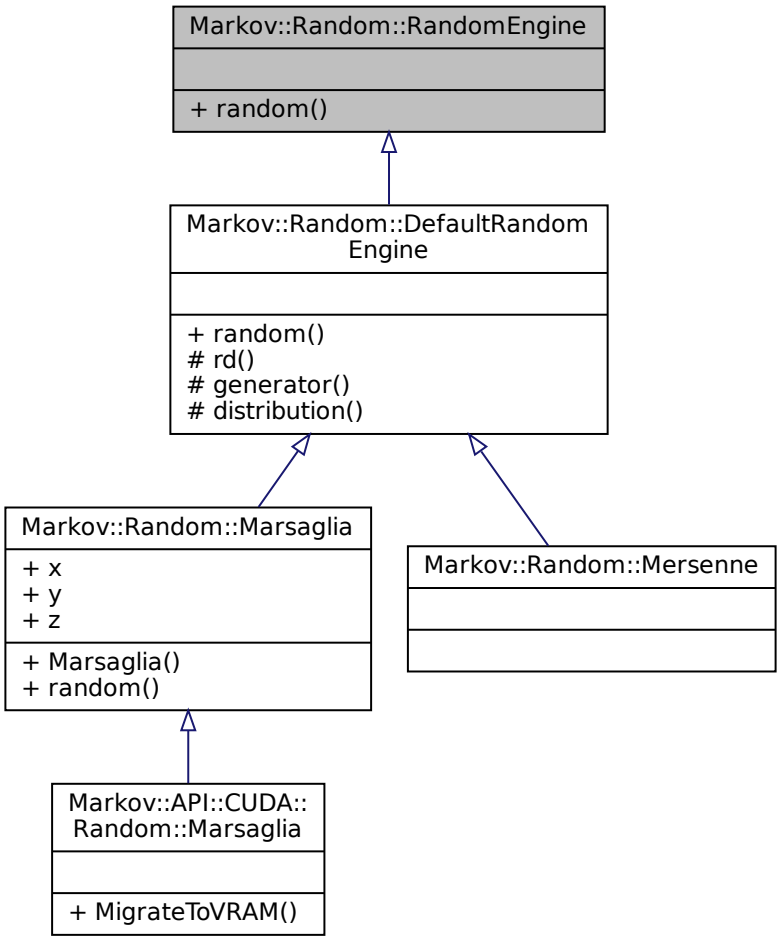
Definition at line 176 of file node.h.

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

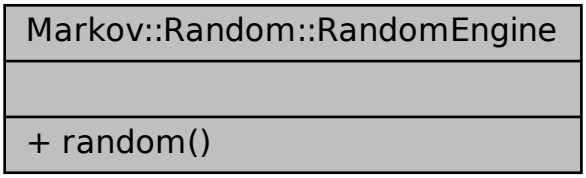
- model.h
- node.h

8.18 Markov::Random::RandomEngine Class Reference

An abstract class for Random Engine.
`#include <random.h>`
Inheritance diagram for Markov::Random::RandomEngine:



Collaboration diagram for Markov::Random::RandomEngine:



Public Member Functions

- virtual unsigned long random ()=0

8.18.1 Detailed Description

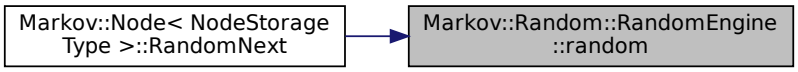
An abstract class for Random Engine.
This class is used for generating random numbers, which are used for random walking on the graph.
Main reason behind allowing different random engines is that some use cases may favor performance, while some favor good random.
Mersenne can be used for truer random, while Marsaglia can be used for deterministic but fast random.
Definition at line 21 of file random.h.

8.18.2 Member Function Documentation

8.18.2.1 random()

```
virtual unsigned long Markov::Random::RandomEngine::random ( )  [inline], [pure virtual]
```

Implemented in Markov::Random::Marsaglia, and Markov::Random::DefaultRandomEngine.
Referenced by Markov::Node< NodeStorageType >::RandomNext().
Here is the caller graph for this function:



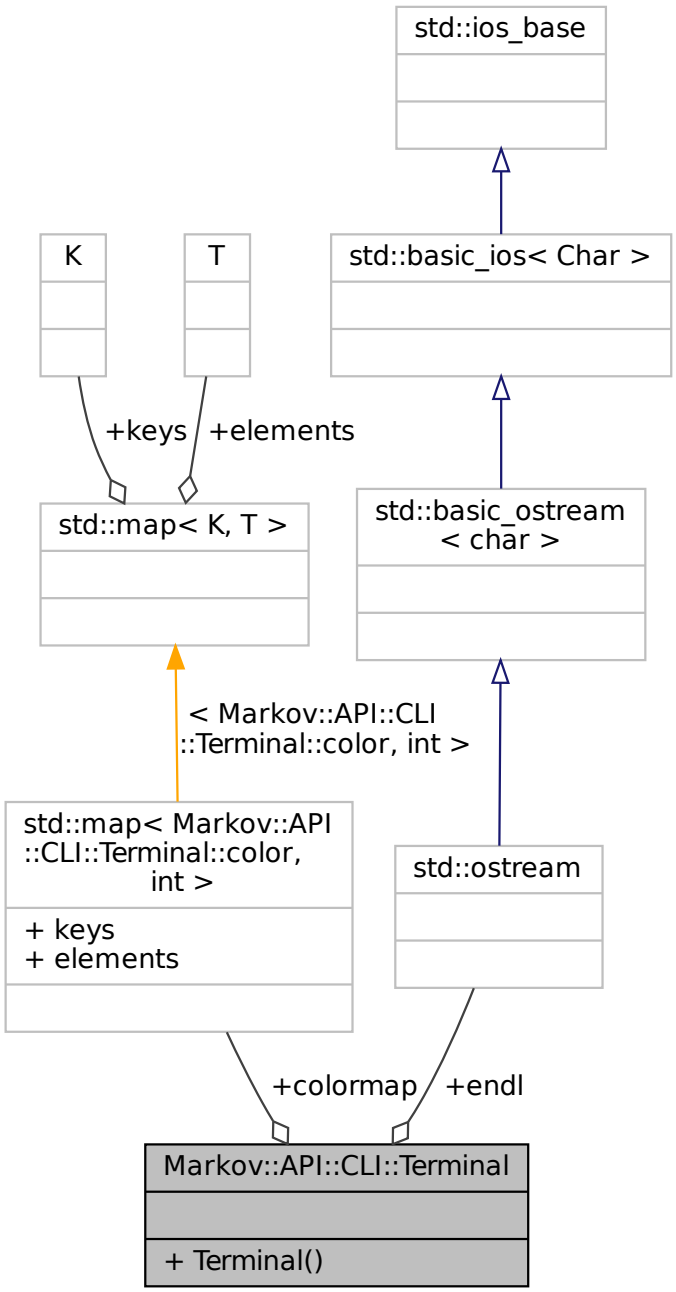
The documentation for this class was generated from the following file:

- random.h

8.19 Markov::API::CLI::Terminal Class Reference

pretty colors for Terminal. Windows Only.
`#include <term.h>`

Collaboration diagram for Markov::API::CLI::Terminal:



Public Types

- enum color {
  RESET, BLACK, RED, GREEN,
  YELLOW, BLUE, MAGENTA, CYAN,
  WHITE, LIGHTGRAY, DARKGRAY, BROWN }

Public Member Functions

- Terminal ()

Static Public Attributes

- static std::map< Markov::API::CLI::Terminal::color, int > colormap
- static std::ostream endl

8.19.1 Detailed Description

pretty colors for Terminal. Windows Only.
Definition at line 18 of file term.h.

8.19.2 Member Enumeration Documentation

8.19.2.1 color

enum Markov::API::CLI::Terminal::color

Enumerator

RESET	
BLACK	
RED	
GREEN	
YELLOW	
BLUE	
MAGENTA	
CYAN	
WHITE	
LIGHTGRAY	
DARKGRAY	
BROWN	

Definition at line 26 of file term.h.

```
00026 { RESET, BLACK, RED, GREEN, YELLOW, BLUE, MAGENTA, CYAN, WHITE, LIGHTGRAY, DARKGRAY, BROWN };
```

8.19.3 Constructor & Destructor Documentation

8.19.3.1 Terminal()

Terminal::Terminal ( )

Default constructor. Get references to stdout and stderr handles.
Definition at line 56 of file term.cpp.

```
00056                    {
00057     /*this->;*/
00058 }
```

8.19.4 Member Data Documentation

8.19.4.1 colormap

```
std::map< Terminal::color, int > Terminal::colormap  [static]
```

Initial value:

```
= {
    {Terminal::color::BLACK, 30},
    {Terminal::color::BLUE, 34},
    {Terminal::color::GREEN, 32},
    {Terminal::color::CYAN, 36},
    {Terminal::color::RED, 31},
    {Terminal::color::MAGENTA, 35},
    {Terminal::color::BROWN, 0},
    {Terminal::color::LIGHTGRAY, 0},
    {Terminal::color::DARKGRAY, 0},
    {Terminal::color::YELLOW, 33},
    {Terminal::color::WHITE, 37},
    {Terminal::color::RESET, 0},
}
```

Definition at line 32 of file term.h.
Referenced by operator<<().

8.19.4.2 endl

```
std::ostream Markov::API::CLI::Terminal::endl  [static]
```

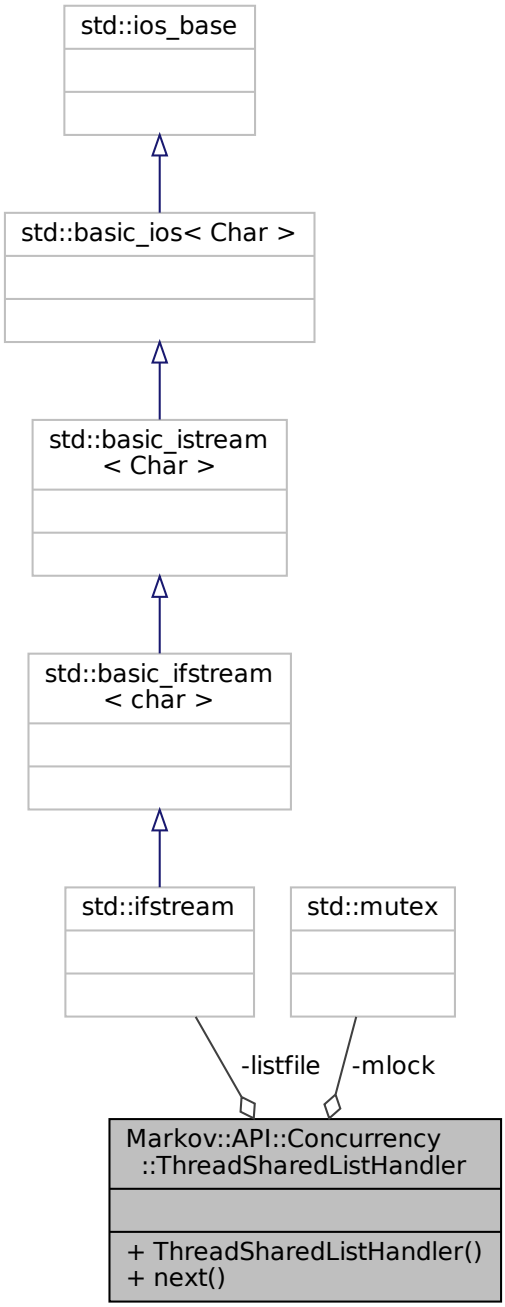
Definition at line 37 of file term.h.
The documentation for this class was generated from the following files:

- term.h
- term.cpp

8.20 Markov::API::Concurrency::ThreadSharedListHandler Class Reference

Simple class for managing shared access to file.
`#include <threadSharedListHandler.h>`

Collaboration diagram for Markov::API::Concurrency::ThreadSharedListHandler:



Public Member Functions

- ThreadSharedListHandler (const char *filename)

  Construct the Thread Handler with a filename.

- bool next (std::string *line)

  Read the next line from the file.

Private Attributes

- std::ifstream listfile
- std::mutex mlock

8.20.1 Detailed Description

Simple class for managing shared access to file.
This class maintains the handover of each line from a file to multiple threads.
When two different threads try to read from the same file while reading a line isn't completed, it can have unexpected results. Line might be split, or might be read twice. This class locks the read action on the list until a line is completed, and then proceeds with the handover.
Definition at line 18 of file threadSharedListHandler.h.

8.20.2 Constructor & Destructor Documentation

8.20.2.1 ThreadSharedListHandler()

```
Markov::API::Concurrency::ThreadSharedListHandler::ThreadSharedListHandler (
            const char * filename )
```

Construct the Thread Handler with a filename.
Simply open the file, and initialize the locks.
Example Use: Simple file read

```
ThreadSharedListHandler listhandler("test.txt");
std::string line;
std::cout « listhandler->next(&line) « "\n";
```

Example Use: Example use case from MarkovPasswords showing multithreaded access

```
void MarkovPasswords::Train(const char* datasetFileName, char delimiter, int threads)   {
     ThreadSharedListHandler listhandler(datasetFileName);
     auto start = std::chrono::high_resolution_clock::now();
     std::vector<std::thread*> threadsV;
     for(int i=0;i<threads;i++){
         threadsV.push_back(new std::thread(&MarkovPasswords::TrainThread, this, &listhandler,
       datasetFileName, delimiter));
     }
    for(int i=0;i<threads;i++){
         threadsV[i]->join();
         delete threadsV[i];
     }
     auto finish = std::chrono::high_resolution_clock::now();
     std::chrono::duration<double> elapsed = finish - start;
     std::cout « "Elapsed time: " « elapsed.count() « " s\n";
 }
 void MarkovPasswords::TrainThread(ThreadSharedListHandler *listhandler, const char* datasetFileName, char
       delimiter){
     char format_str[] ="%ld,%s";
     format_str[2]=delimiter;
     std::string line;
     while (listhandler->next(&line)) {
         long int oc;
         if (line.size() > 100) {
             line = line.substr(0, 100);
         }
         char* linebuf = new char[line.length()+5];
         sscanf_s(line.c_str(), format_str, &oc, linebuf, line.length()+5);
         this->AdjustEdge((const char*)linebuf, oc);
         delete linebuf;
     }
 }
```

Parameters

filename	Filename for the file to manage.

Definition at line 4 of file threadSharedListHandler.cpp.

```
00004                                                                                    {
00005     this->listfile;
00006     this->listfile.open(filename, std::ios_base::binary);
00007 }
```

References listfile.

Referenced by Markov::API::MarkovPasswords::Train().
Here is the caller graph for this function:

8.20.3 Member Function Documentation

8.20.3.1 next()

```
bool Markov::API::Concurrency::ThreadSharedListHandler::next (
            std::string * line )
```

Read the next line from the file.
This action will be blocked until another thread (if any) completes the read operation on the file.
Example Use: Simple file read

```
ThreadSharedListHandler listhandler("test.txt");
std::string line;
std::cout << listhandler->next(&line) << "\n";
```

Definition at line 10 of file threadSharedListHandler.cpp.

```
00010                                                                    {
00011     bool res = false;
00012     this->mlock.lock();
00013     res = (std::getline(this->listfile,*line,'\n'))? true : false;
00014     this->mlock.unlock();
00015
00016     return res;
00017 }
```

References listfile, and mlock.
Referenced by Markov::API::MarkovPasswords::TrainThread().
Here is the caller graph for this function:

8.20.4 Member Data Documentation

8.20.4.1 listfile

```
std::ifstream Markov::API::Concurrency::ThreadSharedListHandler::listfile  [private]
```

Definition at line 88 of file threadSharedListHandler.h.
Referenced by next(), and ThreadSharedListHandler().

8.20.4.2 mlock

```
std::mutex Markov::API::Concurrency::ThreadSharedListHandler::mlock  [private]
```

Definition at line 89 of file threadSharedListHandler.h.
Referenced by next().
The documentation for this class was generated from the following files:

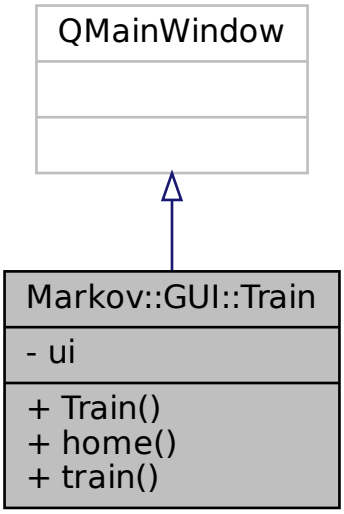
- threadSharedListHandler.h
- threadSharedListHandler.cpp

8.21 Markov::GUI::Train Class Reference

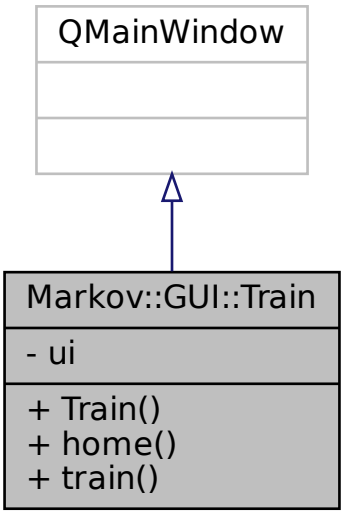
QT Training page class.

`#include <Train.h>`

Inheritance diagram for Markov::GUI::Train:



Collaboration diagram for Markov::GUI::Train:



Public Slots

- void home ()
- void train ()

Public Member Functions

- Train (QWidget *parent=Q_NULLPTR)

Private Attributes

- Ui::Train ui

8.21.1 Detailed Description

QT Training page class.
Definition at line 9 of file Train.h.

8.21.2 Constructor & Destructor Documentation

8.21.2.1 Train()

```
Markov::GUI::Train::Train (
            QWidget * parent = Q_NULLPTR )
```

8.21.3 Member Function Documentation

8.21.3.1 home

```
void Markov::GUI::Train::home ( )  [slot]
```

8.21.3.2 train

```
void Markov::GUI::Train::train ( )  [slot]
```

8.21.4 Member Data Documentation

8.21.4.1 ui

```
Ui::Train Markov::GUI::Train::ui  [private]
```

Definition at line 15 of file Train.h.
The documentation for this class was generated from the following file:

- Train.h

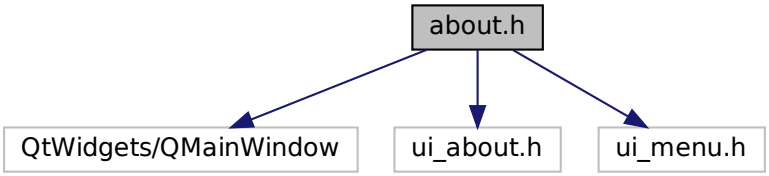
Chapter 9

File Documentation

9.1 about.h File Reference

```
#include <QtWidgets/QMainWindow>
#include "ui_about.h"
#include <ui_menu.h>
```

Include dependency graph for about.h:



Classes

- class Markov::GUI::about

  QT Class for about page.

Namespaces

- Markov

  Namespace for the markov-model related classes. Contains Model, Node and Edge classes.
- Markov::GUI

  namespace for MarkovPasswords API GUI wrapper

9.2 about.h

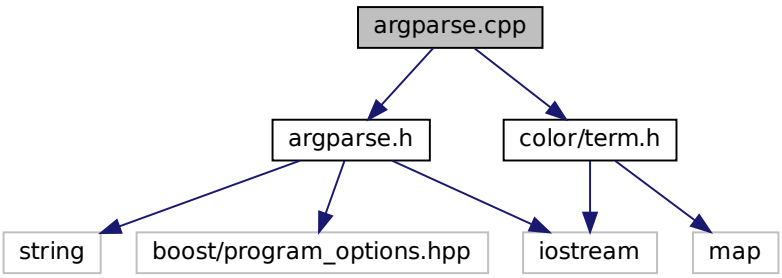
```
00001 #pragma once
00002 #include <QtWidgets/QMainWindow>
00003 #include "ui_about.h"
00004 #include <ui_menu.h>
00005
00006 /** @brief namespace for MarkovPasswords API GUI wrapper
00007 */
00008 namespace Markov::GUI{
00009
00010     /** @brief QT Class for about page
00011     */
00012     class about :public QMainWindow {
```

```
00013     Q_OBJECT
00014     public:
00015         about(QWidget* parent = Q_NULLPTR);
00016
00017     private:
00018         Ui:: main ui;
00019
00020
00021     };
00022 };
```

9.3 argparse.cpp File Reference

```
#include "argparse.h"
#include "color/term.h"
```

Include dependency graph for argparse.cpp:



9.4 argparse.cpp

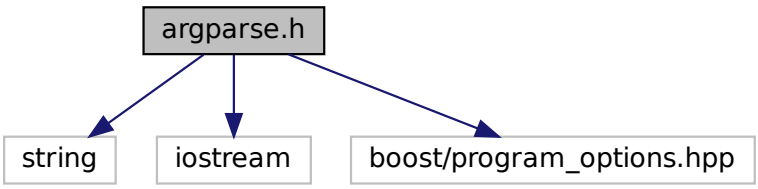
```
00001 #include "argparse.h"
00002 #include "color/term.h"
00003
00004 Markov::API::CLI::ProgramOptions* Markov::API::CLI::Argparse::parse(int argc, char** argv) { return 0;
       }
00005
00006
00007
00008 void Markov::API::CLI::Argparse::help() {
00009     std::cout «
00010     "Markov Passwords - Help\n"
00011     "Options:\n"
00012     "   \n"
00013     "   -of --outputfilename\n"
00014     "       Filename to output the generation results\n"
00015     "   -ef --exportfilename\n"
00016     "       filename to export built model to\n"
00017     "   -if --importfilename\n"
00018     "       filename to import model from\n"
00019     "   -n (generate count)\n"
00020     "       Number of lines to generate\n"
00021     "   \n"
00022     "Usage: \n"
00023     "   markov.exe -if empty_model.mdl -ef model.mdl\n"
00024     "       import empty_model.mdl and train it with data from stdin. When done, output the model to
      model.mdl\n"
00025     "\n"
00026     "   markov.exe -if empty_model.mdl -n 15000 -of wordlist.txt\n"
00027     "       import empty_model.mdl and generate 15000 words to wordlist.txt\n"
00028
00029      « std::endl;
00030 }
```

9.5 argparse.h File Reference

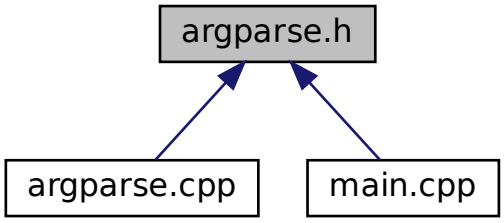
```
#include <string>
```

```
#include <iostream>
#include <boost/program_options.hpp>
```

Include dependency graph for argparse.h:



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



Classes

- struct Markov::API::CLI::_programOptions

  Structure to hold parsed cli arguements.

- class Markov::API::CLI::Argparse

  Parse command line arguements.

Namespaces

- Markov

  Namespace for the markov-model related classes. Contains Model, Node and Edge classes.

- Markov::API

  Namespace for the MarkovPasswords API.

- Markov::API::CLI

  Structure to hold parsed cli arguements.

Macros

- #define BOOST_ALL_DYN_LINK 1

Typedefs

- typedef struct Markov::API::CLI::_programOptions Markov::API::CLI::ProgramOptions

  Structure to hold parsed cli arguements.

9.5.1 Macro Definition Documentation

9.5.1.1 BOOST_ALL_DYN_LINK

```
#define BOOST_ALL_DYN_LINK 1
```

Definition at line 4 of file argparse.h.

9.6 argparse.h

```
00001 #include<string>
00002 #include<iostream>
00003
00004 #define BOOST_ALL_DYN_LINK 1
00005
00006 #include <boost/program_options.hpp>
00007 /** @brief Structure to hold parsed cli arguements.
00008 */
00009 namespace opt = boost::program_options;
00010
00011 /**
00012     @brief Namespace for the CLI objects
00013 */
00014 namespace Markov::API::CLI{
00015
00016     /** @brief Structure to hold parsed cli arguements. */
00017     typedef struct _programOptions {
00018         /**
00019             @brief Import flag to validate import
00020         */
00021         bool bImport;
00022
00023         /**
00024             @brief Export flag to validate export
00025         */
00026         bool bExport;
00027
00028         /**
00029             @brief Failure flag to validate succesfull running
00030         */
00031         bool bFailure;
00032
00033         /**
00034             @brief Seperator character to use with training data. (character between occurence and
    value)"
00035         */
00036         char seperator;
00037
00038         /**
00039             @brief Import name of our model
00040         */
00041         std::string importname;
00042
00043         /**
00044             @brief Import name of our given wordlist
00045         */
00046         std::string exportname;
00047
00048         /**
00049             @brief Import name of our given wordlist
00050         */
00051         std::string wordlistname;
00052
00053         /**
00054             @brief Output name of our generated password list
00055         */
00056         std::string outputfilename;
00057
00058         /**
00059             @brief The name of the given dataset
00060         */
00061         std::string datasetname;
00062
```

```
00063         /**
00064             @brief Number of passwords to be generated
00065         */
00066         int generateN;
00067
00068     } ProgramOptions;
00069
00070
00071     /** @brief Parse command line arguements
00072     */
00073     class Argparse {
00074     public:
00075
00076         Argparse();
00077
00078         /** @brief Parse command line arguements.
00079          *
00080          * Parses command line arguements to populate ProgramOptions structure.
00081          *
00082          * @param argc Number of command line arguements
00083          * @param argv Array of command line parameters
00084         */
00085         Argparse(int argc, char** argv) {
00086
00087             /*bool bImp;
00088             bool bExp;
00089             bool bFail;
00090             char sprt;
00091             std::string imports;
00092             std::string exports;
00093             std::string outputs;
00094             std::string datasets;
00095             int generateN;
00096             */
00097             opt::options_description desc("Options");
00098
00099
00100             desc.add_options()
00101                 ("generate", "Generate strings with given parameters")
00102                 ("train", "Train model with given parameters")
00103                 ("combine", "Combine")
00104                 ("import", opt::value<std::string>(), "Import model file")
00105                 ("output", opt::value<std::string>(), "Output model file. This model will be exported
      when done. Will be ignored for generation mode")
00106                 ("dataset", opt::value<std::string>(), "Dataset file to read input from training. Will
      be ignored for generation mode")
00107                 ("seperator", opt::value<char>(), "Seperator character to use with training data.
      (character between occurence and value)")
00108                 ("wordlist", opt::value<std::string>(), "Wordlist file path to export generation
      results to. Will be ignored for training mode")
00109                 ("count", opt::value<int>(), "Number of lines to generate. Ignored in training mode")
00110                 ("verbosity", "Output verbosity")
00111                 ("help", "Option definitions");
00112
00113                 opt::variables_map vm;
00114
00115                 opt::store(opt::parse_command_line(argc, argv, desc), vm);
00116
00117                 opt::notify(vm);
00118
00119                 //std::cout << desc << std::endl;
00120                 if (vm.count("help")) {
00121                 std::cout << desc << std::endl;
00122                 }
00123
00124                 if (vm.count("output") == 0) this->po.outputfilename = "NULL";
00125                 else if (vm.count("output") == 1) {
00126                     this->po.outputfilename = vm["output"].as<std::string>();
00127                     this->po.bExport = true;
00128                 }
00129                 else {
00130                     this->po.bFailure = true;
00131                     std::cout << "UNIDENTIFIED INPUT" << std::endl;
00132                     std::cout << desc << std::endl;
00133                 }
00134
00135
00136                 if (vm.count("dataset") == 0) this->po.datasetname = "NULL";
00137                 else if (vm.count("dataset") == 1) {
00138                     this->po.datasetname = vm["dataset"].as<std::string>();
00139                 }
00140                 else {
00141                     this->po.bFailure = true;
00142                     std::cout << "UNIDENTIFIED INPUT" << std::endl;
00143                     std::cout << desc << std::endl;
00144                 }
00145
```

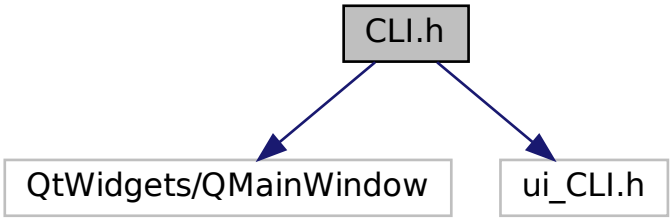
```
00146
00147                 if (vm.count("wordlist") == 0) this->po.wordlistname = "NULL";
00148                 else if (vm.count("wordlist") == 1) {
00149                     this->po.wordlistname = vm["wordlist"].as<std::string>();
00150                 }
00151                 else {
00152                     this->po.bFailure = true;
00153                     std::cout « "UNIDENTIFIED INPUT" « std::endl;
00154                     std::cout « desc « std::endl;
00155                 }
00156
00157                 if (vm.count("import") == 0) this->po.importname = "NULL";
00158                 else if (vm.count("import") == 1) {
00159                     this->po.importname = vm["import"].as<std::string>();
00160                     this->po.bImport = true;
00161                 }
00162                 else {
00163                     this->po.bFailure = true;
00164                     std::cout « "UNIDENTIFIED INPUT" « std::endl;
00165                     std::cout « desc « std::endl;
00166                 }
00167
00168
00169                 if (vm.count("count") == 0) this->po.generateN = 0;
00170                 else if (vm.count("count") == 1) {
00171                     this->po.generateN = vm["count"].as<int>();
00172                 }
00173                 else {
00174                     this->po.bFailure = true;
00175                     std::cout « "UNIDENTIFIED INPUT" « std::endl;
00176                     std::cout « desc « std::endl;
00177                 }
00178
00179                 /*std::cout « vm["output"].as<std::string>() « std::endl;
00180                 std::cout « vm["dataset"].as<std::string>() « std::endl;
00181                 std::cout « vm["wordlist"].as<std::string>() « std::endl;
00182                 std::cout « vm["output"].as<std::string>() « std::endl;
00183                 std::cout « vm["count"].as<int>() « std::endl;*/
00184
00185
00186                 //else if (vm.count("train")) std::cout « "train oldu" « std::endl;
00187         }
00188
00189         /** @brief Getter for command line options
00190          *
00191          * Getter for ProgramOptions populated by the arguement parser
00192          * @returns ProgramOptions structure.
00193         */
00194         Markov::API::CLI::ProgramOptions getProgramOptions(void) {
00195             return this->po;
00196         }
00197
00198         /** @brief Initialize program options structure.
00199          *
00200          * @param i boolean, true if import operation is flagged
00201          * @param e boolean, true if export operation is flagged
00202          * @param bf boolean, true if there is something wrong with the command line parameters
00203          * @param s seperator character for the import function
00204          * @param iName import filename
00205          * @param exName export filename
00206          * @param oName output filename
00207          * @param dName corpus filename
00208          * @param n number of passwords to be generated
00209          *
00210         */
00211         void setProgramOptions(bool i, bool e, bool bf, char s, std::string iName, std::string exName,
      std::string oName, std::string dName, int n) {
00212             this->po.bImport = i;
00213             this->po.bExport = e;
00214             this->po.seperator = s;
00215             this->po.bFailure = bf;
00216             this->po.generateN = n;
00217             this->po.importname = iName;
00218             this->po.exportname = exName;
00219             this->po.outputfilename = oName;
00220             this->po.datasetname = dName;
00221
00222             /*strcpy_s(this->po.importname,256,iName);
00223             strcpy_s(this->po.exportname,256,exName);
00224             strcpy_s(this->po.outputfilename,256,oName);
00225             strcpy_s(this->po.datasetname,256,dName);*/
00226
00227         }
00228
00229         /** @brief parse cli commands and return
00230         * @param argc - Program arguement count
00231         * @param argv - Program arguement values array
```

```
00232         * @return ProgramOptions structure.
00233         */
00234         static Markov::API::CLI::ProgramOptions* parse(int argc, char** argv);
00235
00236
00237         /** @brief Print help string.
00238         */
00239         static void help();
00240
00241     private:
00242         /**
00243             @brief ProgramOptions structure object
00244         */
00245
00246         Markov::API::CLI::ProgramOptions po;
00247     };
00248
00249 };
```

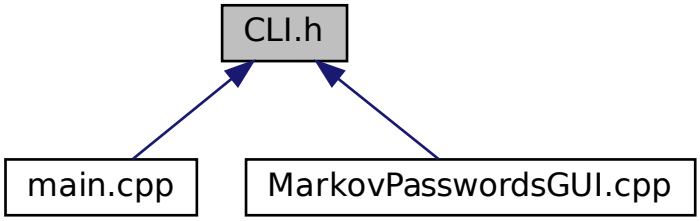
9.7 CLI.h File Reference

```
#include <QtWidgets/QMainWindow>
#include "ui_CLI.h"
```

Include dependency graph for CLI.h:



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



Classes

- class Markov::GUI::CLI

  QT CLI Class.

Namespaces

- Markov

  Namespace for the markov-model related classes. Contains Model, Node and Edge classes.

- Markov::GUI

  namespace for MarkovPasswords API GUI wrapper

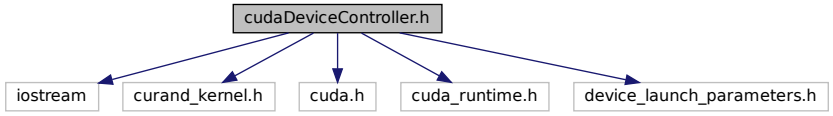
9.8 CLI.h

```
00001 #pragma once
00002 #include <QtWidgets/QMainWindow>
00003 #include "ui_CLI.h"
00004
00005 namespace Markov::GUI{
00006     /** @brief QT CLI Class
00007     */
00008     class CLI :public QMainWindow {
00009         Q_OBJECT
00010     public:
00011         CLI(QWidget* parent = Q_NULLPTR);
00012
00013     private:
00014         Ui::CLI ui;
00015
00016     public slots:
00017         void start();
00018         void statistics();
00019         void about();
00020     };
00021 };
```

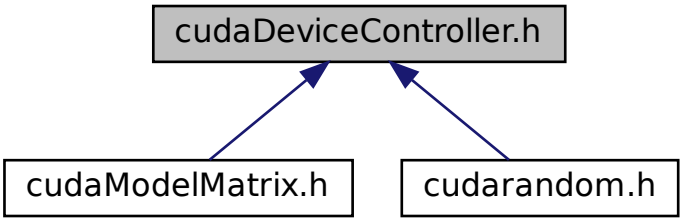
9.9 cudaDeviceController.h File Reference

```
#include <iostream>
#include <curand_kernel.h>
#include <cuda.h>
#include <cuda_runtime.h>
#include <device_launch_parameters.h>
```

Include dependency graph for cudaDeviceController.h:



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



Classes

- class Markov::API::CUDA::CUDADeviceController

  Controller class for CUDA device.

Namespaces

- Markov

  Namespace for the markov-model related classes. Contains Model, Node and Edge classes.
- Markov::API

  Namespace for the MarkovPasswords API.
- Markov::API::CUDA

  Namespace for objects requiring CUDA libraries.

9.10 cudaDeviceController.h

```
00001
00002 #pragma once
00003 #include <iostream>
00004 #include <curand_kernel.h>
00005 #include <cuda.h>
00006 #include <cuda_runtime.h>
00007 #include <device_launch_parameters.h>
00008
00009 /** @brief Namespace for objects requiring CUDA libraries.
00010 */
00011 namespace Markov::API::CUDA{
00012     /** @brief Controller class for CUDA device
00013      *
00014      * This implementation only supports Nvidia devices.
00015     */
00016     class CUDADeviceController{
00017     public:
00018         /** @brief List CUDA devices in the system.
00019          *
00020          * This function will print details of every CUDA capable device in the system.
00021          *
00022          * @b Example @b output:
00023          * @code{.txt}
00024          * Device Number: 0
00025          * Device name: GeForce RTX 2070
00026          * Memory Clock Rate (KHz): 7001000
00027          * Memory Bus Width (bits): 256
00028          * Peak Memory Bandwidth (GB/s): 448.064
00029          * Max Linear Threads: 1024
00030          * @endcode
00031         */
00032         __host__ static void ListCudaDevices();
00033
00034     protected:
00035         /** @brief Check results of the last operation on GPU.
```

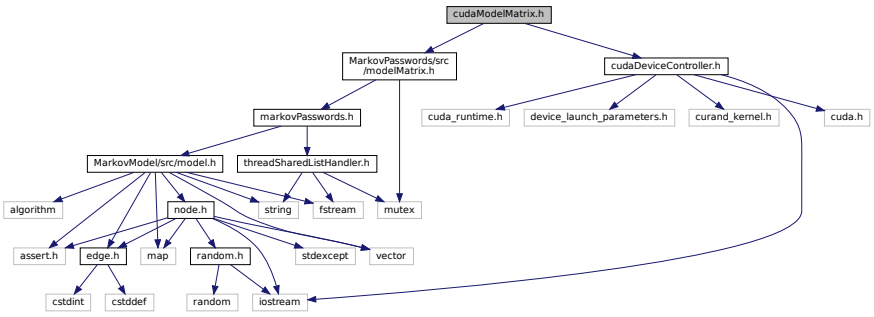
```
00036          *
00037          * Check the status returned from cudaMalloc/cudaMemcpy to find failures.
00038          *
00039          * If a failure occurs, its assumed beyond redemption, and exited.
00040          * @param _status Cuda error status to check
00041          * @param msg Message to print in case of a failure
00042          * @return 0 if successful, 1 if failure.
00043          * @b Example @b output:
00044          * @code{.cpp}
00045          * char *da, a = "test";
00046          * cudastatus = cudaMalloc((char **)&da, 5*sizeof(char*));
00047          * CudaCheckNotifyErr(cudastatus, "Failed to allocate VRAM for *da.\n");
00048          * @endcode
00049         */
00050         __host__ static int CudaCheckNotifyErr(cudaError_t _status, const char* msg, bool bExit=true);
00051
00052
00053         /** @brief Malloc a 2D array in device space
00054          *
00055          * This function will allocate enough space on VRAM for flattened 2D array.
00056          *
00057          * @param dst destination pointer
00058          * @param row row size of the 2d array
00059          * @param col column size of the 2d array
00060          * @return cudaError_t status of the cudaMalloc operation
00061          *
00062          * @b Example @b output:
00063          * @code{.cpp}
00064          *   cudaError_t cudastatus;
00065          *   char* dst;
00066          *   cudastatus = CudaMalloc2DToFlat<char>(&dst, 5, 15);
00067          *   if(cudastatus!=cudaSuccess){
00068          *       CudaCheckNotifyErr(cudastatus, "  CudaMalloc2DToFlat Failed.", false);
00069          *   }
00070          * @endcode
00071         */
00072         template <typename T>
00073         __host__ static cudaError_t CudaMalloc2DToFlat(T** dst, int row, int col){
00074             cudaError_t cudastatus = cudaMalloc((T **)dst, row*col*sizeof(T));
00075             CudaCheckNotifyErr(cudastatus, "cudaMalloc Failed.", false);
00076             return cudastatus;
00077         }
00078
00079
00080         /** @brief Memcpy a 2D array in device space after flattening
00081          *
00082          * Resulting buffer will not be true 2D array.
00083          *
00084          * @param dst destination pointer
00085          * @param rc  source pointer
00086          * @param row row size of the 2d array
00087          * @param col column size of the 2d array
00088          * @return cudaError_t status of the cudaMalloc operation
00089          *
00090          * @b Example @b output:
00091          * @code{.cpp}
00092          *   cudaError_t cudastatus;
00093          *   char* dst;
00094          *   cudastatus = CudaMalloc2DToFlat<char>(&dst, 5, 15);
00095          *   CudaCheckNotifyErr(cudastatus, "  CudaMalloc2DToFlat Failed.", false);
00096          *   cudastatus = CudaMemcpy2DToFlat<char>(*dst,src,15,15);
00097          *   CudaCheckNotifyErr(cudastatus, "  CudaMemcpy2DToFlat Failed.", false);
00098          * @endcode
00099         */
00100        template <typename T>
00101        __host__ static cudaError_t CudaMemcpy2DToFlat(T* dst, T** src, int row, int col){
00102            T* tempbuf = new T[row*col];
00103            for(int i=0;i<row;i++){
00104                memcpy(&(tempbuf[row*i]), src[i], col);
00105            }
00106            return cudaMemcpy(dst, tempbuf, row*col*sizeof(T), cudaMemcpyHostToDevice);
00107
00108        }
00109
00110        /** @brief Both malloc and memcpy a 2D array into device VRAM.
00111          *
00112          * Resulting buffer will not be true 2D array.
00113          *
00114          * @param dst destination pointer
00115          * @param rc  source pointer
00116          * @param row row size of the 2d array
00117          * @param col column size of the 2d array
00118          * @return cudaError_t status of the cudaMalloc operation
00119          *
00120          * @b Example @b output:
00121          * @code{.cpp}
00122          *   cudaError_t cudastatus;
```

```
00123          *   char* dst;
00124          *   cudastatus = CudaMigrate2DFlat<long int>(
00125          *      &dst, this->valueMatrix, this->matrixSize, this->matrixSize);
00126          *   CudaCheckNotifyErr(cudastatus, "    Cuda failed to initialize value matrix row.");
00127          * @endcode
00128         */
00129         template <typename T>
00130         __host__ static cudaError_t CudaMigrate2DFlat(T** dst, T** src, int row, int col){
00131             cudaError_t cudastatus;
00132             cudastatus = CudaMalloc2DToFlat<T>(dst, row, col);
00133             if(cudastatus!=cudaSuccess){
00134                 CudaCheckNotifyErr(cudastatus, "  CudaMalloc2DToFlat Failed.", false);
00135                 return cudastatus;
00136             }
00137             cudastatus = CudaMemcpy2DToFlat<T>(*dst,src,row,col);
00138             CudaCheckNotifyErr(cudastatus, "  CudaMemcpy2DToFlat Failed.", false);
00139             return cudastatus;
00140         }
00141
00142
00143     private:
00144     };
00145 };
```

9.11 cudaModelMatrix.h File Reference

```
#include "MarkovPasswords/src/modelMatrix.h"
#include "cudaDeviceController.h"
```

Include dependency graph for cudaModelMatrix.h:



Classes

- class Markov::API::CUDA::CUDAModelMatrix

  Extension of Markov::API::ModelMatrix which is modified to run on GPU devices.

Namespaces

- Markov

  Namespace for the markov-model related classes. Contains Model, Node and Edge classes.
- Markov::API

  Namespace for the MarkovPasswords API.
- Markov::API::CUDA

  Namespace for objects requiring CUDA libraries.

Functions

- __global__ void Markov::API::CUDA::FastRandomWalkCUDAKernel (unsigned long int n, int minLen, int maxLen, char *outputBuffer, char *matrixIndex, long int *totalEdgeWeights, long int *valueMatrix, char *edgeMatrix, int matrixSize, int memoryPerKernelGrid, unsigned long *seed)

  CUDA kernel for the FastRandomWalk operation.

- __device__ char * Markov::API::CUDA::strchr (char *p, char c, int s_len)

  *srtchr implementation on **device** space*

9.12 cudaModelMatrix.h

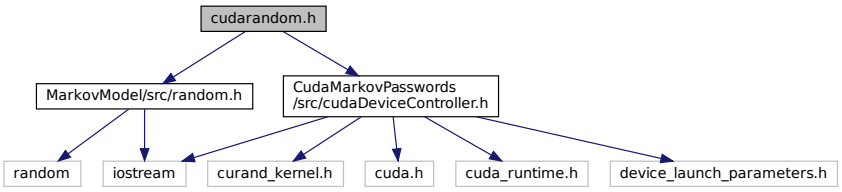
```
00001 #include "MarkovPasswords/src/modelMatrix.h"
00002 #include "cudaDeviceController.h"
00003
00004 /** @brief Namespace for objects requiring CUDA libraries.
00005 */
00006 namespace Markov::API::CUDA{
00007     /** @brief Extension of Markov::API::ModelMatrix which is modified to run on GPU devices.
00008      *
00009      * This implementation only supports Nvidia devices.
00010     */
00011     class CUDAModelMatrix : public ModelMatrix, public CUDADeviceController{
00012     public:
00013
00014         /** @brief Migrate the class members to the VRAM
00015          *
00016          * Cannot be used without calling Markov::API::ModelMatrix::ConstructMatrix at least once.
00017          * This function will manage the memory allocation and data transfer from CPU RAM to GPU VRAM.
00018          *
00019          * Newly allocated VRAM pointers are set in the class member variables.
00020          *
00021         */
00022         __host__ void MigrateMatrix();
00023
00024         /** @brief Flatten migrated matrix from 2d to 1d
00025          *
00026          *
00027         */
00028         __host__ void FlattenMatrix();
00029
00030         /** @brief Random walk on the Matrix-reduced Markov::Model
00031          *
00032          * TODO
00033          *
00034          *
00035          * @param n - Number of passwords to generate.
00036          * @param wordlistFileName - Filename to write to
00037          * @param minLen - Minimum password length to generate
00038          * @param maxLen - Maximum password length to generate
00039          * @param threads - number of OS threads to spawn
00040          * @param bFileIO - If false, filename will be ignored and will output to stdout.
00041          *
00042          *
00043          * @code{.cpp}
00044          * Markov::API::ModelMatrix mp;
00045          * mp.Import("models/finished.mdl");
00046          * mp.FastRandomWalk(50000000,"./wordlist.txt",6,12,25, true);
00047          * @endcode
00048          *
00049         */
00050         __host__ void FastRandomWalk(unsigned long int n, const char* wordlistFileName, int minLen,
      int maxLen, bool bFileIO);
00051
00052     protected:
00053
00054         /** @brief Allocate the output buffer for kernel operation
00055          *
00056          * TODO
00057          *
00058          *
00059          * @param n - Number of passwords to generate.
00060          * @param singleGenMaxLen - maximum string length for a single generation
00061          * @param CUDAKernelGridSize - Total number of grid members in CUDA kernel
00062          * @param sizePerGrid - Size to allocate per grid member
00063          * @return pointer to the allocation on VRAM
00064          *
00065          *
00066         */
00067         __host__ char* AllocVRAMOutputBuffer(long int n, long int singleGenMaxLen, long int
      CUDAKernelGridSize,long int sizePerGrid);
00068     private:
00069
00070         /**
00071             @brief VRAM Address pointer of edge matrix (from modelMatrix.h)
00072         */
00073         char* device_edgeMatrix;
00074
00075         /**
00076             @brief VRAM Address pointer of value matrix (from modelMatrix.h)
00077         */
```

```
00078         long int *device_valueMatrix;
00079
00080         /**
00081             @brief VRAM Address pointer of matrixIndex (from modelMatrix.h)
00082         */
00083         char *device_matrixIndex;
00084
00085         /**
00086             @brief VRAM Address pointer of total edge weights (from modelMatrix.h)
00087         */
00088         long int *device_totalEdgeWeights;
00089
00090
00091         /**
00092             @brief RandomWalk results in device
00093         */
00094         char* device_outputBuffer;
00095
00096         /**
00097             @brief RandomWalk  results in host
00098         */
00099         char* outputBuffer;
00100
00101         /**
00102             @brief Adding Edge matrix end-to-end and resize to 1-D array for better perfomance on
      traversing
00103         */
00104         char* flatEdgeMatrix;
00105
00106         /**
00107             @brief Adding Value matrix end-to-end and resize to 1-D array for better perfomance on
      traversing
00108         */
00109         long int* flatValueMatrix;
00110
00111     };
00112
00113     /** @brief CUDA kernel for the FastRandomWalk operation
00114      *
00115      * Will be initiated by CPU and continued by GPU (__global__ tag)
00116      *
00117      *
00118      * @param n - Number of passwords to generate.
00119      * @param minlen - minimum string length for a single generation
00120      * @param maxLen - maximum string length for a single generation
00121      * @param outputBuffer - VRAM ptr to the output buffer
00122      * @param matrixIndex - VRAM ptr to the matrix indices
00123      * @param totalEdgeWeights - VRAM ptr to the totalEdgeWeights array
00124      * @param valueMatrix - VRAM ptr to the edge weights array
00125      * @param edgeMatrix - VRAM ptr to the edge representations array
00126      * @param matrixSize - Size of the matrix dimensions
00127      * @param memoryPerKernelGrid - Maximum memory usage per kernel grid
00128      * @param seed - seed chunk to generate the random from (generated & used by Marsaglia)
00129      *
00130      *
00131      *
00132     */
00133     __global__ void FastRandomWalkCUDAKernel(unsigned long int n, int minLen, int maxLen, char*
      outputBuffer,
00134         char* matrixIndex, long int* totalEdgeWeights, long int* valueMatrix, char *edgeMatrix,
00135         int matrixSize, int memoryPerKernelGrid, unsigned long *seed);//, unsigned long mex, unsigned
      long mey, unsigned long mez);
00136
00137
00138     /** @brief srtchr implementation on __device__ space
00139      *
00140      * Fint the first matching index of a string
00141      *
00142      *
00143      * @param p - string to check
00144      * @param c - character to match
00145      * @param s_len - maximum string length
00146      * @returns pointer to the match
00147     */
00148     __device__ char* strchr(char* p, char c, int s_len);
00149
00150 };
```

9.13 cudarandom.h File Reference

```
#include "MarkovModel/src/random.h"
#include "CudaMarkovPasswords/src/cudaDeviceController.h"
```

Include dependency graph for cudarandom.h:



Classes

- class Markov::API::CUDA::Random::Marsaglia

  *Extension of Markov::Random::Marsaglia which is capable o working on **device** space.*

Namespaces

- Markov

  Namespace for the markov-model related classes. Contains Model, Node and Edge classes.
- Markov::API

  Namespace for the MarkovPasswords API.
- Markov::API::CUDA

  Namespace for objects requiring CUDA libraries.
- Markov::API::CUDA::Random

  *Namespace for Random engines operable under **device** space.*

Functions

- __device__ unsigned long Markov::API::CUDA::Random::devrandom (unsigned long &x, unsigned long &y, unsigned long &z)

  *Marsaglia Random Generation function operable in **device** space.*

9.14 cudarandom.h

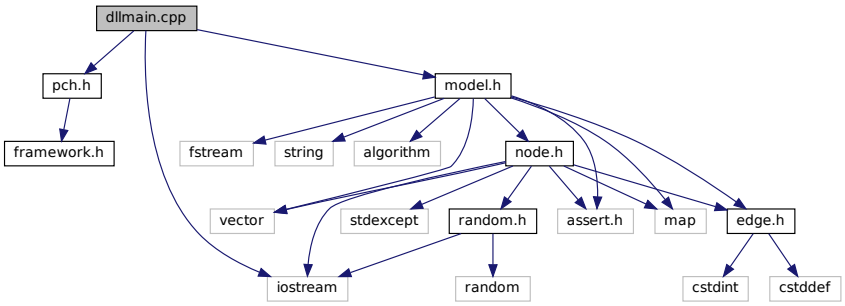
```
00001 #pragma once
00002 #include "MarkovModel/src/random.h"
00003 #include "CudaMarkovPasswords/src/cudaDeviceController.h"
00004
00005 /** @brief Namespace for Random engines operable under __device__ space.
00006 */
00007 namespace Markov::API::CUDA::Random{
00008
00009     /** @brief Extension of Markov::Random::Marsaglia which is capable o working on __device__ space.
00010     */
00011     class Marsaglia : public Markov::Random::Marsaglia, public CUDADeviceController{
00012     public:
00013
00014         /** @brief Migrate a Marsaglia[] to VRAM as seedChunk
00015          * @param MEarr Array of Marsaglia Engines
00016          * @param gridSize GridSize of the CUDA Kernel, aka size of array
00017          * @returns pointer to the resulting seed chunk in device VRAM.
00018         */
00019         static unsigned long* MigrateToVRAM(Markov::API::CUDA::Random::Marsaglia *MEarr, long int
      gridSize){
00020             cudaError_t cudastatus;
00021             unsigned long* seedChunk;
00022             cudastatus = cudaMalloc((unsigned long**)&seedChunk, gridSize*3*sizeof(unsigned long));
00023             CudaCheckNotifyErr(cudastatus, "Failed to allocate seed buffer");
00024             unsigned long *temp = new unsigned long[gridSize*3];
00025             for(int i=0;i<gridSize;i++){
00026                 temp[i*3]   = MEarr[i].x;
00027                 temp[i*3+1] = MEarr[i].y;
00028                 temp[i*3+2] = MEarr[i].z;
```

```
00029             }
00030             //for(int i=0;i<gridSize*3;i++) std::cout « temp[i] « "\n";
00031             cudaMemcpy(seedChunk, temp, gridSize*3*sizeof(unsigned long), cudaMemcpyHostToDevice);
00032             CudaCheckNotifyErr(cudastatus, "Failed to memcpy seed buffer.");
00033             return seedChunk;
00034         }
00035     };
00036
00037     /** @brief Marsaglia Random Generation function operable in __device__ space
00038      * @param x marsaglia internal x. Not constant, (ref)
00039      * @param y marsaglia internal y. Not constant, (ref)
00040      * @param z marsaglia internal z. Not constant, (ref)
00041      * @returns returns z
00042     */
00043     __device__ unsigned long devrandom(unsigned long &x, unsigned long &y, unsigned long &z){
00044         unsigned long t;
00045         x ^= x « 16;
00046         x ^= x » 5;
00047         x ^= x « 1;
00048
00049         t = x;
00050         x = y;
00051         y = z;
00052         z = t ^ x ^ y;
00053
00054         return z;
00055     }
00056 };
```

9.15 dllmain.cpp File Reference

```
#include "pch.h"
#include "model.h"
#include <iostream>
```

Include dependency graph for dllmain.cpp:



9.16 dllmain.cpp

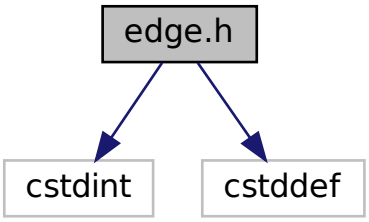
```
00001 #include "pch.h"
00002 #include "model.h"
00003 #include <iostream>
00004
00005
00006 #ifdef _WIN32
00007 __declspec(dllexport) void dll_loadtest() {
00008     std::cout « "External function called.\n";
00009     //cudaTestEntry();
00010 }
00011
00012 BOOL APIENTRY DllMain(HMODULE hModule, DWORD ul_reason_for_call, LPVOID lpReserved)
00013 {
00014     switch (ul_reason_for_call)
00015     {
00016     case DLL_PROCESS_ATTACH:
00017     case DLL_THREAD_ATTACH:
00018     case DLL_THREAD_DETACH:
00019     case DLL_PROCESS_DETACH:
00020         break;
00021     }
```

```
00022     return TRUE;
00023 }
00024
00025 #endif
```

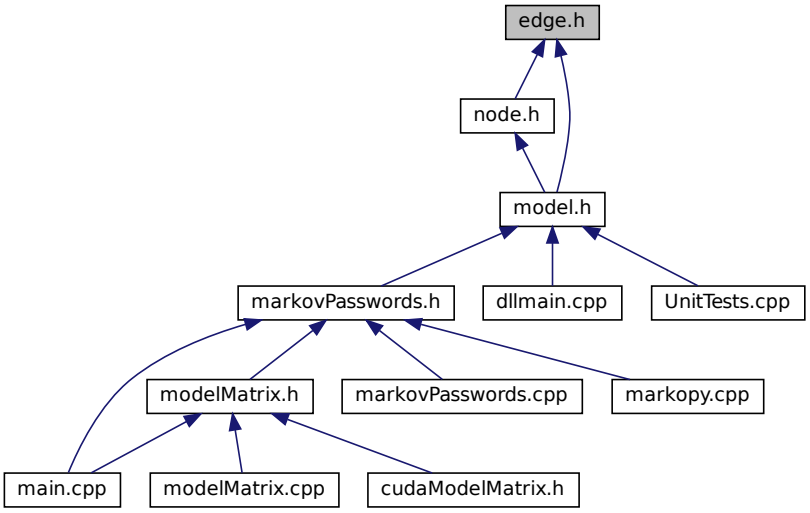
9.17 edge.h File Reference

```
#include <cstdint>
#include <cstddef>
```

Include dependency graph for edge.h:



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



Classes

- class Markov::Node< storageType >

  A node class that for the vertices of model. Connected with eachother using Edge.

- class Markov::Edge< NodeStorageType >

  Edge class used to link nodes in the model together.

Namespaces

- Markov

  Namespace for the markov-model related classes. Contains Model, Node and Edge classes.

9.18 edge.h

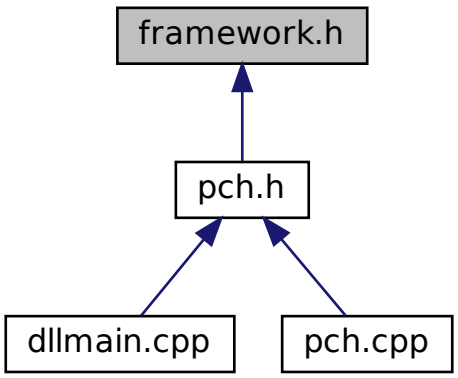
```
00001 #pragma once
00002 #include <cstdint>
00003 #include <cstddef>
00004
00005 namespace Markov {
00006
00007     template <typename NodeStorageType>
00008     class Node;
00009
00010     /** @brief Edge class used to link nodes in the model together.
00011     *
00012     Has LeftNode, RightNode, and EdgeWeight of the edge.
00013     Edges are *UNIDIRECTIONAL* in this model. They can only be traversed LeftNode to RightNode.
00014     */
00015     template <typename NodeStorageType>
00016     class Edge {
00017     public:
00018
00019         /** @brief Default constructor.
00020         */
00021         Edge<NodeStorageType>();
00022
00023         /** @brief Constructor. Initialize edge with given RightNode and LeftNode
00024          * @param _left - Left node of this edge.
00025          * @param _right - Right node of this edge.
00026          *
00027          * @b Example @b Use: Construct edge
00028          * @code{.cpp}
00029          * Markov::Node<unsigned char>* src = new Markov::Node<unsigned char>('a');
00030          * Markov::Node<unsigned char>* target1 = new Markov::Node<unsigned char>('b');
00031          * Markov::Edge<unsigned char>* e1 = new Markov::Edge<unsigned char>(src, target1);
00032          * @endcode
00033          *
00034         */
00035         Edge<NodeStorageType>(Node<NodeStorageType>* _left, Node<NodeStorageType>* _right);
00036
00037         /** @brief Adjust the edge EdgeWeight with offset.
00038          * Adds the offset parameter to the edge EdgeWeight.
00039          * @param offset - NodeValue to be added to the EdgeWeight
00040          *
00041          * @b Example @b Use: Construct edge
00042          * @code{.cpp}
00043          * Markov::Node<unsigned char>* src = new Markov::Node<unsigned char>('a');
00044          * Markov::Node<unsigned char>* target1 = new Markov::Node<unsigned char>('b');
00045          * Markov::Edge<unsigned char>* e1 = new Markov::Edge<unsigned char>(src, target1);
00046          *
00047          * e1->AdjustEdge(25);
00048          *
00049          * @endcode
00050         */
00051         void AdjustEdge(long int offset);
00052
00053         /** @brief Traverse this edge to RightNode.
00054          * @return Right node. If this is a terminator node, return NULL
00055          *
00056          *
00057          * @b Example @b Use: Traverse a node
00058          * @code{.cpp}
00059          * Markov::Node<unsigned char>* src = new Markov::Node<unsigned char>('a');
00060          * Markov::Node<unsigned char>* target1 = new Markov::Node<unsigned char>('b');
00061          * Markov::Edge<unsigned char>* e1 = new Markov::Edge<unsigned char>(src, target1);
00062          *
00063          * e1->AdjustEdge(25);
00064          * Markov::Edge<unsigned char>* e2 = e1->traverseNode();
00065          * @endcode
00066          *
00067         */
00068         inline Node<NodeStorageType>* TraverseNode();
00069
00070         /** @brief Set LeftNode of this edge.
00071         * @param node - Node to be linked with.
00072         */
00073         void SetLeftEdge (Node<NodeStorageType>*);
00074         /** @brief Set RightNode of this edge.
00075         * @param node - Node to be linked with.
00076         */
```

```
00077         void SetRightEdge(Node<NodeStorageType>*);
00078
00079         /** @brief return edge's EdgeWeight.
00080         * @return edge's EdgeWeight.
00081         */
00082         inline uint64_t EdgeWeight();
00083
00084         /** @brief return edge's LeftNode
00085         * @return edge's LeftNode.
00086         */
00087         Node<NodeStorageType>* LeftNode();
00088
00089         /** @brief return edge's RightNode
00090         * @return edge's RightNode.
00091         */
00092         inline Node<NodeStorageType>* RightNode();
00093
00094     private:
00095         /**
00096             @brief source node
00097         */
00098         Node<NodeStorageType>* _left;
00099
00100         /**
00101             @brief target node
00102         */
00103         Node<NodeStorageType>* _right;
00104
00105         /** @brief
00106             Edge Edge Weight
00107         */
00108         long int _weight;
00109     };
00110
00111
00112 };
00113
00114 //default constructor of edge
00115 template <typename NodeStorageType>
00116 Markov::Edge<NodeStorageType>::Edge() {
00117     this->_left = NULL;
00118     this->_right = NULL;
00119     this->_weight = 0;
00120 }
00121 //constructor of edge
00122 template <typename NodeStorageType>
00123 Markov::Edge<NodeStorageType>::Edge(Markov::Node<NodeStorageType>* _left,
       Markov::Node<NodeStorageType>* _right) {
00124     this->_left = _left;
00125     this->_right = _right;
00126     this->_weight = 0;
00127 }
00128 //to AdjustEdge the edges by the edge with its offset
00129 template <typename NodeStorageType>
00130 void Markov::Edge<NodeStorageType>::AdjustEdge(long int offset) {
00131     this->_weight += offset;
00132     this->LeftNode()->UpdateTotalVerticeWeight(offset);
00133 }
00134 //to TraverseNode the node
00135 template <typename NodeStorageType>
00136 inline Markov::Node<NodeStorageType>* Markov::Edge<NodeStorageType>::TraverseNode() {
00137     if (this->RightNode()->NodeValue() == 0xff) //terminator node
00138         return NULL;
00139     return _right;
00140 }
00141 //to set the LeftNode of the node
00142 template <typename NodeStorageType>
00143 void Markov::Edge<NodeStorageType>::SetLeftEdge(Markov::Node<NodeStorageType>* n) {
00144     this->_left = n;
00145 }
00146 //to set the RightNode of the node
00147 template <typename NodeStorageType>
00148 void Markov::Edge<NodeStorageType>::SetRightEdge(Markov::Node<NodeStorageType>* n) {
00149     this->_right = n;
00150 }
00151 //to get the EdgeWeight of the node
00152 template <typename NodeStorageType>
00153 inline uint64_t Markov::Edge<NodeStorageType>::EdgeWeight() {
00154     return this->_weight;
00155 }
00156 //to get the LeftNode of the node
00157 template <typename NodeStorageType>
00158 Markov::Node<NodeStorageType>* Markov::Edge<NodeStorageType>::LeftNode() {
00159     return this->_left;
00160 }
00161 //to get the RightNode of the node
00162 template <typename NodeStorageType>
```

```
00163 inline Markov::Node<NodeStorageType>* Markov::Edge<NodeStorageType>::RightNode() {
00164     return this->_right;
00165 }
```

9.19 framework.h File Reference

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



Macros

- #define WIN32_LEAN_AND_MEAN

9.19.1 Macro Definition Documentation

9.19.1.1 WIN32_LEAN_AND_MEAN

```
#define WIN32_LEAN_AND_MEAN
```

Definition at line 3 of file framework.h.

9.20 framework.h

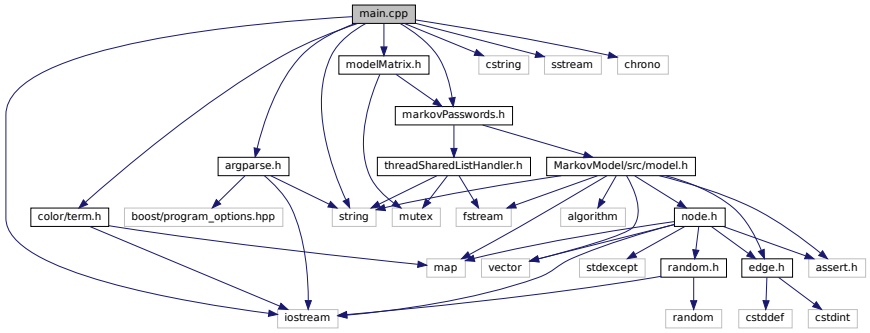
```
00001 #pragma once
00002
00003 #define WIN32_LEAN_AND_MEAN             // Exclude rarely-used stuff from Windows headers
00004 // Windows Header Files
00005
00006 #ifdef _WIN32
00007 #include <windows.h>
00008 #endif
```

9.21 main.cpp File Reference

```
#include <iostream>
#include "color/term.h"
#include "argparse.h"
#include <string>
#include <cstring>
```

```
#include <sstream>
#include "markovPasswords.h"
#include "modelMatrix.h"
#include <chrono>
```

Include dependency graph for src/main.cpp:



Functions

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

  Launch CLI tool.

9.21.1 Function Documentation

9.21.1.1 main()

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

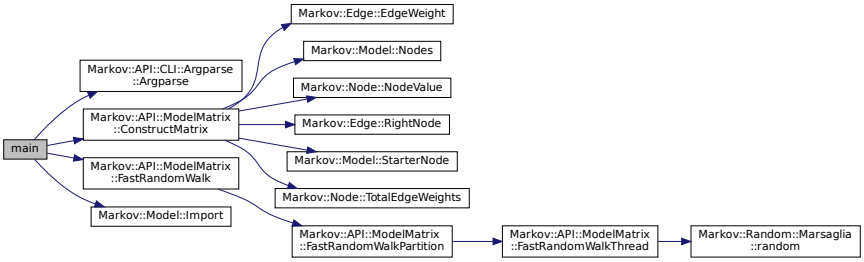
Launch CLI tool.

Definition at line 14 of file src/main.cpp.

```
00014                               {
00015
00016     Markov::API::CLI::Terminal t;
00017     /*
00018     ProgramOptions* p  = Argparse::parse(argc, argv);
00019
00020     if (p==0 || p->bFailure) {
00021         std::cout « TERM_FAIL « "Arguments Failed to Parse" « std::endl;
00022         Argparse::help();
00023     }*/
00024     Markov::API::CLI::Argparse a(argc,argv);
00025
00026     Markov::API::ModelMatrix markovPass;
00027     std::cerr « "Importing model.\n";
00028     markovPass.Import("models/finished.mdl");
00029     std::cerr « "Import done. \n";
00030     markovPass.ConstructMatrix();
00031     std::chrono::steady_clock::time_point begin = std::chrono::steady_clock::now();
00032     //markovPass.FastRandomWalk(50000000,"/media/ignis/Stuff/wordlist.txt",6,12,25, true);
00033     markovPass.FastRandomWalk(500000000,"/media/ignis/Stuff/wordlist2.txt",6,12,25, true);
00034     std::chrono::steady_clock::time_point end = std::chrono::steady_clock::now();
00035
00036     std::cerr « "Finished in:" « std::chrono::duration_cast<std::chrono::milliseconds> (end -
      begin).count() « " milliseconds" « std::endl;
00037     return 0;
00038 }
```

References Markov::API::CLI::Argparse::Argparse(), Markov::API::ModelMatrix::ConstructMatrix(), Markov::API::ModelMatrix::FastRa
and Markov::Model< NodeStorageType >::Import().

Here is the call graph for this function:



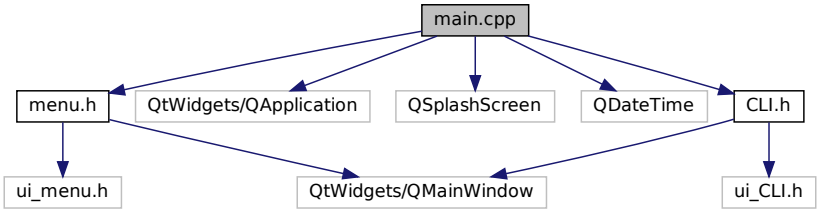
9.22 src/main.cpp

```
00001 #pragma once
00002 #include <iostream>
00003 #include "color/term.h"
00004 #include "argparse.h"
00005 #include <string>
00006 #include <cstring>
00007 #include <sstream>
00008 #include "markovPasswords.h"
00009 #include "modelMatrix.h"
00010 #include <chrono>
00011
00012 /** @brief Launch CLI tool.
00013 */
00014 int main(int argc, char** argv) {
00015
00016     Markov::API::CLI::Terminal t;
00017     /*
00018     ProgramOptions* p  = Argparse::parse(argc, argv);
00019
00020     if (p==0 || p->bFailure) {
00021         std::cout << TERM_FAIL << "Arguments Failed to Parse" << std::endl;
00022         Argparse::help();
00023     }*/
00024     Markov::API::CLI::Argparse a(argc,argv);
00025
00026     Markov::API::ModelMatrix markovPass;
00027     std::cerr << "Importing model.\n";
00028     markovPass.Import("models/finished.mdl");
00029     std::cerr << "Import done. \n";
00030     markovPass.ConstructMatrix();
00031     std::chrono::steady_clock::time_point begin = std::chrono::steady_clock::now();
00032     //markovPass.FastRandomWalk(50000000,"/media/ignis/Stuff/wordlist.txt",6,12,25, true);
00033     markovPass.FastRandomWalk(500000000,"/media/ignis/Stuff/wordlist2.txt",6,12,25, true);
00034     std::chrono::steady_clock::time_point end = std::chrono::steady_clock::now();
00035
00036     std::cerr << "Finished in:" << std::chrono::duration_cast<std::chrono::milliseconds> (end -
       begin).count() << " milliseconds" << std::endl;
00037     return 0;
00038 }
```

9.23 main.cpp File Reference

```
#include "menu.h"
#include <QtWidgets/QApplication>
#include <QSplashScreen>
#include <QDateTime>
#include "CLI.h"
```

Include dependency graph for UI/src/main.cpp:



Functions

- int main (int argc, char ∗argv[ ])

  Launch UI.

9.23.1 Function Documentation

9.23.1.1 main()

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

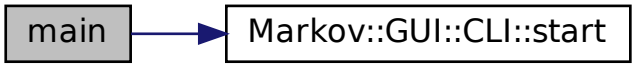
Launch UI.

Definition at line 12 of file UI/src/main.cpp.

```
00013 {
00014
00015
00016
00017     QApplication a(argc, argv);
00018
00019     QPixmap loadingPix("views/startup.jpg");
00020     QSplashScreen splash(loadingPix);
00021     splash.show();
00022     QDateTime time = QDateTime::currentDateTime();
00023     QDateTime currentTime = QDateTime::currentDateTime();   //Record current time
00024     while (time.secsTo(currentTime) <= 5)                   //5 is the number of seconds to delay
00025     {
00026         currentTime = QDateTime::currentDateTime();
00027         a.processEvents();
00028     };
00029
00030
00031     CLI w;
00032     w.show();
00033     splash.finish(&w);
00034     return a.exec();
00035 }
```

References Markov::GUI::CLI::start().

Here is the call graph for this function:



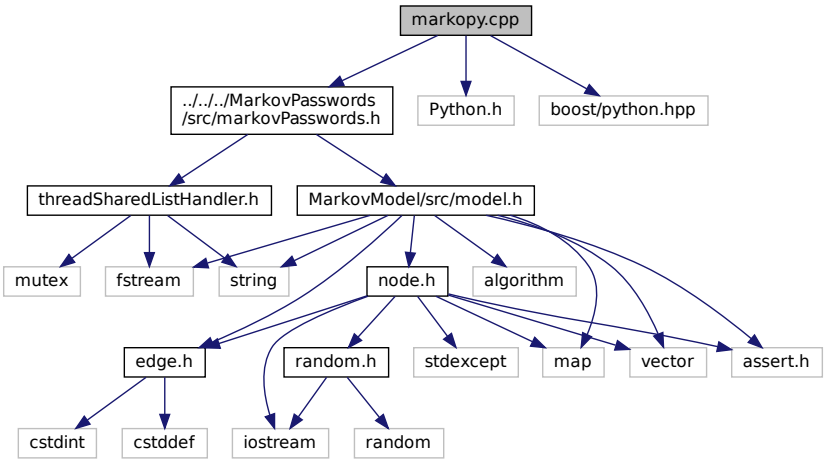
9.24 UI/src/main.cpp

```
00001 //#include "MarkovPasswordsGUI.h"
00002 #include "menu.h"
00003 #include <QtWidgets/QApplication>
00004 #include <QSplashScreen>
00005 #include < QDateTime >
00006 #include "CLI.h"
00007
00008 using namespace Markov::GUI;
00009
00010 /** @brief Launch UI.
00011 */
00012 int main(int argc, char *argv[])
00013 {
00014
00015
00016
00017     QApplication a(argc, argv);
00018
00019     QPixmap loadingPix("views/startup.jpg");
00020     QSplashScreen splash(loadingPix);
00021     splash.show();
00022     QDateTime time = QDateTime::currentDateTime();
00023     QDateTime currentTime = QDateTime::currentDateTime();   //Record current time
00024     while (time.secsTo(currentTime) <= 5)                   //5 is the number of seconds to delay
00025     {
00026         currentTime = QDateTime::currentDateTime();
00027         a.processEvents();
00028     };
00029
00030
00031     CLI w;
00032     w.show();
00033     splash.finish(&w);
00034     return a.exec();
00035 }
```

9.25 markopy.cpp File Reference

```
#include "../../../MarkovPasswords/src/markovPasswords.h"
#include <Python.h>
#include <boost/python.hpp>
```

Include dependency graph for markopy.cpp:



Namespaces

- Markov

  Namespace for the markov-model related classes. Contains Model, Node and Edge classes.
- Markov::Markopy

Macros

- #define BOOST_PYTHON_STATIC_LIB

Functions

- Markov::Markopy::BOOST_PYTHON_MODULE (markopy)

9.25.1 Macro Definition Documentation

9.25.1.1 BOOST_PYTHON_STATIC_LIB

```
#define BOOST_PYTHON_STATIC_LIB
```

Definition at line 4 of file markopy.cpp.

9.26 markopy.cpp

```
00001 #pragma once
00002 #include "../../../MarkovPasswords/src/markovPasswords.h"
00003
00004 #define BOOST_PYTHON_STATIC_LIB
00005 #include <Python.h>
00006 #include <boost/python.hpp>
00007
00008 using namespace boost::python;
00009
00010 namespace Markov::Markopy{
00011     BOOST_PYTHON_MODULE(markopy)
00012     {
00013         bool (Markov::API::MarkovPasswords::*Import)(const char*) = &Markov::Model<char>::Import;
00014         bool (Markov::API::MarkovPasswords::*Export)(const char*) = &Markov::Model<char>::Export;
00015         class_<Markov::API::MarkovPasswords>("MarkovPasswords", init<>())
00016             .def(init<>())
00017             .def("Train", &Markov::API::MarkovPasswords::Train,
00018             "Train the model\n"
00019             "\n"
00020             ":param datasetFileName: Ifstream* to the dataset. If null, use class member\n"
00021             ":param delimiter: a character, same as the delimiter in dataset content\n"
00022             ":param threads: number of OS threads to spawn\n")
00023             .def("Generate", &Markov::API::MarkovPasswords::Generate,
00024             "Generate passwords from a trained model.\n"
00025             ":param n: Ifstream* to the dataset. If null, use class member\n"
00026             ":param wordlistFileName: a character, same as the delimiter in dataset content\n"
00027             ":param minLen: number of OS threads to spawn\n"
00028             ":param maxLen: Ifstream* to the dataset. If null, use class member\n"
00029             ":param threads: a character, same as the delimiter in dataset content\n"
00030             ":param threads: number of OS threads to spawn\n")
00031             .def("Import", Import, "Import a model file.")
00032             .def("Export", Export, "Export a model to file.")
00033         ;
00034     };
00035 };
```

9.27 markopy_cli.py File Reference

Namespaces

- markopy_cli

Functions

- def markopy_cli.cli_init (input_model)
- def markopy_cli.cli_train (model, dataset, seperator, output, output_forced=False, bulk=False)
- def markopy_cli.cli_generate (model, wordlist, bulk=False)

Variables

- markopy_cli.parser
- markopy_cli.help
- markopy_cli.default
- markopy_cli.action
- markopy_cli.args = parser.parse_args()
- markopy_cli.corpus_list = os.listdir(args.dataset)
- def markopy_cli.model = cli_init(args.input)
- markopy_cli.output_file_name = corpus
- string markopy_cli.model_extension = ""
- markopy_cli.output_forced
- markopy_cli.True
- markopy_cli.bulk
- markopy_cli.model_list = os.listdir(args.input)
- markopy_cli.model_base = input
- markopy_cli.output

9.28 markopy_cli.py

```
00001 #!/usr/bin/python3
00002 """
00003   @namespace Markov::Markopy::Python
00004 """
00005
00006 import markopy
00007 import argparse
00008 import allogate as logging
00009 import re
00010 import os
00011
00012 parser = argparse.ArgumentParser(description="Python wrapper for MarkovPasswords.",
00013                                  epilog=f"""Sample runs:
00014 {__file__} train untrained.mdl -d dataset.dat -s "\\t" -o trained.mdl
00015     Import untrained.mdl, train it with dataset.dat which has tab delimited data, output resulting
      model to trained.mdl\n
00016
00017 {__file__} generate trained.mdl -n 500 -w output.txt
00018     Import trained.mdl, and generate 500 lines to output.txt
00019
00020 {__file__} combine untrained.mdl -d dataset.dat -s "\\t" -n 500 -w output.txt
00021     Train and immediately generate 500 lines to output.txt. Do not export trained model.
00022
00023 {__file__} combine untrained.mdl -d dataset.dat -s "\\t" -n 500 -w output.txt -o trained.mdl
00024     Train and immediately generate 500 lines to output.txt. Export trained model.
00025 """, formatter_class=argparse.RawTextHelpFormatter)
00026
00027 parser.add_argument("mode", help="Operation mode, supported modes: \"generate\", \"train\" and
      \"combine\".")
00028 parser.add_argument("input", help="Input model file. This model will be imported before starting
      operation.\n"
00029                                   + "For more information on the file structure for input, check out
      the wiki page.")
00030 parser.add_argument("-o", "--output",
00031                     help="Output model filename. This model will be exported when done. Will be
      ignored for generation mode.")
00032 parser.add_argument("-d", "--dataset",
00033                     help="Dataset filename to read input from for training. Will be ignored for
      generation mode.\n"
00034                     + "Dataset is occurrence of a string and the string value seperated by a
      seperator. For more info "
00035                     + "on the dataset file structure, check out the github wiki page.")
00036 parser.add_argument("-s", "--seperator",
00037                     help="Seperator character to use with training data.(character between occurrence
      and value)\n"
00038                     + "For more information on dataset/corpus file structure, check out the github
      wiki.")
00039 parser.add_argument("-w", "--wordlist",
00040                     help="Wordlist filename path to export generation results to. Will be ignored for
      training mode")
00041 parser.add_argument("--min", default=6, help="Minimum length that is allowed during generation.\n"
00042                     + "Any string shorter than this paremeter will retry to continue instead of
      proceeding to "
00043                     + "finishing node")
00044 parser.add_argument("--max", default=12, help="Maximum length that is allowed during generation.\n"
```

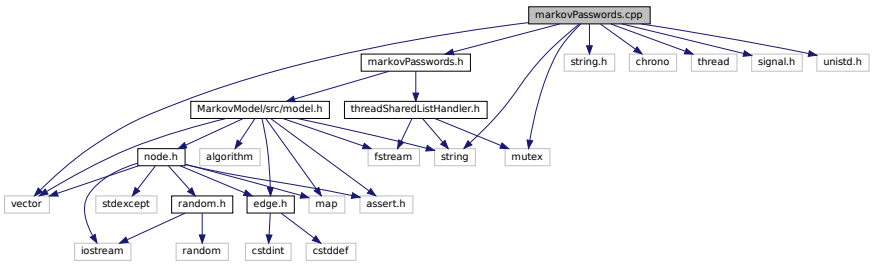
```
00045                     +"Any string that does reaches this length are cut off irregardless to their
      position on the model.")
00046 parser.add_argument("-n", "--count", help="Number of lines to generate. Ignored in training mode.")
00047 parser.add_argument("-t", "--threads", default=10, help="Number of threads to use with
      training/generation.\n"
00048                     +"This many OS threads will be created for training/generation functions")
00049 parser.add_argument("-v", "--verbosity", action="count", help="Output verbosity.\n"
00050                     + "Set verbosity to 1: -v\n"
00051                     + "Set verbosity to 3: -vvv\n"
00052                     + "Print pretty much everything, including caller functions: -vvvvvvvvvvvvvv")
00053 parser.add_argument("-b", "--bulk", action="store_true",
00054                     help="Bulk generate or bulk train every corpus/model in the folder.\n"
00055                     + "If working on this mode, output/input/dataset parameters should be a folder.\n"
00056                     + "Selected operation (generate/train) will be applied to each file in the folder,
      and "
00057                     + "output to the output directory.")
00058 args = parser.parse_args()
00059
00060
00061 def cli_init(input_model):
00062     logging.VERBOSITY = 0
00063     if args.verbosity:
00064         logging.VERBOSITY = args.verbosity
00065         logging.pprint(f"Verbosity set to {args.verbosity}.", 2)
00066
00067     logging.pprint("Initializing model.", 1)
00068     model = markopy.MarkovPasswords()
00069     logging.pprint("Model initialized.", 2)
00070
00071     logging.pprint("Importing model file.", 1)
00072
00073     if (not os.path.isfile(input_model)):
00074         logging.pprint(f"Model file at {input_model} not found. Check the file path, or working
      directory")
00075         exit(1)
00076
00077     model.Import(input_model)
00078     logging.pprint("Model imported successfully.", 2)
00079     return model
00080
00081
00082 def cli_train(model, dataset, seperator, output, output_forced=False, bulk=False):
00083     if not (dataset and seperator and (output or not output_forced)):
00084         logging.pprint(
00085             f"Training mode requires -d/--dataset{', -o/--output' if output_forced else ''} and
      -s/--seperator parameters. Exiting.")
00086         exit(2)
00087
00088     if (not bulk and not os.path.isfile(dataset)):
00089         logging.pprint(f"{dataset} doesn't exists. Check the file path, or working directory")
00090         exit(3)
00091
00092     if (output and os.path.isfile(output)):
00093         logging.pprint(f"{output} exists and will be overwritten.", 1)
00094
00095     if (seperator == '\\t'):
00096         logging.pprint("Escaping seperator.", 3)
00097         seperator = '\t'
00098
00099     if (len(seperator) != 1):
00100         logging.pprint(f'Delimiter must be a single character, and "{seperator}" is not accepted.')
00101         exit(4)
00102
00103     logging.pprint(f'Starting training.', 3)
00104     model.Train(dataset, seperator, int(args.threads))
00105     logging.pprint(f'Training completed.', 2)
00106
00107     if (output):
00108         logging.pprint(f'Exporting model to {output}', 2)
00109         model.Export(output)
00110     else:
00111         logging.pprint(f'Model will not be exported.', 1)
00112
00113
00114 def cli_generate(model, wordlist, bulk=False):
00115     if not (wordlist or args.count):
00116         logging.pprint("Generation mode requires -w/--wordlist and -n/--count parameters. Exiting.")
00117         exit(2)
00118
00119     if (bulk and os.path.isfile(wordlist)):
00120         logging.pprint(f"{wordlist} exists and will be overwritten.", 1)
00121     model.Generate(int(args.count), wordlist, int(args.min), int(args.max), int(args.threads))
00122
00123
00124 if (args.bulk):
00125     logging.pprint(f"Bulk mode operation chosen.", 4)
00126
```

```
00127     if (args.mode.lower() == "train"):
00128         if (os.path.isdir(args.output) and not os.path.isfile(args.output)) and (
00129                 os.path.isdir(args.dataset) and not os.path.isfile(args.dataset)):
00130             corpus_list = os.listdir(args.dataset)
00131             for corpus in corpus_list:
00132                 model = cli_init(args.input)
00133                 logging.pprint(f"Training {args.input} with {corpus}", 2)
00134                 output_file_name = corpus
00135                 model_extension = ""
00136                 if "." in args.input:
00137                     model_extension = args.input.split(".")[-1]
00138                 cli_train(model, f"{args.dataset}/{corpus}", args.seperator,
00139                           f"{args.output}/{corpus}.{model_extension}", output_forced=True, bulk=True)
00140         else:
00141             logging.pprint("In bulk training, output and dataset should be a directory.")
00142             exit(1)
00143
00144     elif (args.mode.lower() == "generate"):
00145         if (os.path.isdir(args.wordlist) and not os.path.isfile(args.wordlist)) and (
00146                 os.path.isdir(args.input) and not os.path.isfile(args.input)):
00147             model_list = os.listdir(args.input)
00148             print(model_list)
00149             for input in model_list:
00150                 logging.pprint(f"Generating from {args.input}/{input} to {args.wordlist}/{input}.txt",
      2)
00151
00152                 model = cli_init(f"{args.input}/{input}")
00153                 model_base = input
00154                 if "." in args.input:
00155                     model_base = input.split(".")[1]
00156                 cli_generate(model, f"{args.wordlist}/{model_base}.txt", bulk=True)
00157         else:
00158             logging.pprint("In bulk generation, input and wordlist should be directory.")
00159
00160 else:
00161     model = cli_init(args.input)
00162     if (args.mode.lower() == "generate"):
00163         cli_generate(model, args.wordlist)
00164
00165
00166     elif (args.mode.lower() == "train"):
00167         cli_train(model, args.dataset, args.seperator, args.output, output_forced=True)
00168
00169
00170     elif (args.mode.lower() == "combine"):
00171         cli_train(model, args.dataset, args.seperator, args.output)
00172         cli_generate(model, args.wordlist)
00173
00174
00175     else:
00176         logging.pprint("Invalid mode arguement given.")
00177         logging.pprint("Accepted modes: 'Generate', 'Train', 'Combine'")
00178         exit(5)
```

9.29 markovPasswords.cpp File Reference

```
#include "markovPasswords.h"
#include <string.h>
#include <chrono>
#include <thread>
#include <vector>
#include <mutex>
#include <string>
#include <signal.h>
#include <unistd.h>
```

Include dependency graph for markovPasswords.cpp:



Functions

- void intHandler (int dummy)

Variables

- static volatile int keepRunning = 1

9.29.1 Function Documentation

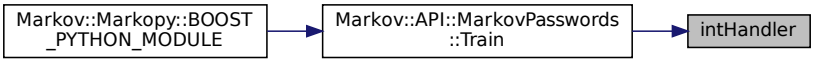
9.29.1.1 intHandler()

```
void intHandler (
            int dummy )
```

Definition at line 18 of file markovPasswords.cpp.

```
00018                           {
00019     std::cout « "You wanted this man by presing CTRL-C ! Ok bye.";
00020     //Sleep(5000);
00021     keepRunning = 0;
00022     exit(0);
00023 }
```

References keepRunning.
Referenced by Markov::API::MarkovPasswords::Train().
Here is the caller graph for this function:



9.29.2 Variable Documentation

9.29.2.1 keepRunning

```
volatile int keepRunning = 1  [static]
```

Definition at line 16 of file markovPasswords.cpp.
Referenced by intHandler(), and Markov::API::MarkovPasswords::TrainThread().

9.30 markovPasswords.cpp

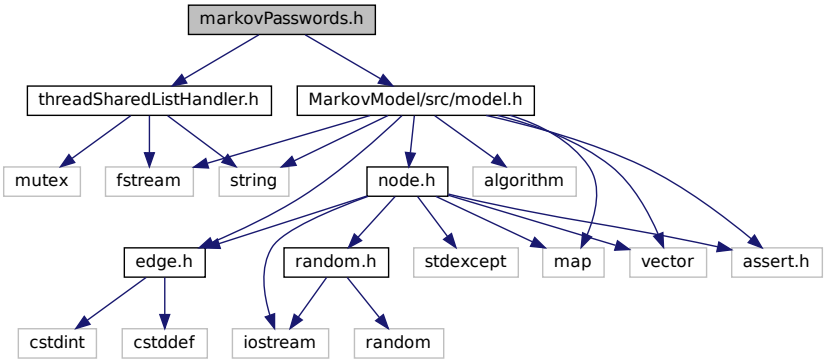
```
00001 #pragma once
00002 #include "markovPasswords.h"
00003 #include <string.h>
00004 #include <chrono>
00005 #include <thread>
00006 #include <vector>
00007 #include <mutex>
00008 #include <string>
00009 #include <signal.h>
00010 #ifdef _WIN32
00011 #include <Windows.h>
00012 #else
00013 #include <unistd.h>
00014 #endif
00015
00016 static volatile int keepRunning = 1;
00017
00018 void intHandler(int dummy) {
00019     std::cout « "You wanted this man by presing CTRL-C ! Ok bye.";
00020     //Sleep(5000);
00021     keepRunning = 0;
00022     exit(0);
00023 }
00024
00025
00026 Markov::API::MarkovPasswords::MarkovPasswords() : Markov::Model<char>(){
00027
00028
00029 }
00030
00031 Markov::API::MarkovPasswords::MarkovPasswords(const char* filename) {
00032
00033     std::ifstream* importFile;
00034
00035     this->Import(filename);
00036
00037     //std::ifstream* newFile(filename);
00038
00039     //importFile = newFile;
00040
00041 }
00042
00043 std::ifstream* Markov::API::MarkovPasswords::OpenDatasetFile(const char* filename){
00044
00045     std::ifstream* datasetFile;
00046
00047     std::ifstream newFile(filename);
00048
00049     datasetFile = &newFile;
00050
00051     this->Import(datasetFile);
00052     return datasetFile;
00053 }
00054
00055
00056
00057 void Markov::API::MarkovPasswords::Train(const char* datasetFileName, char delimiter, int threads)   {
00058   signal(SIGINT, intHandler);
00059     Markov::API::Concurrency::ThreadSharedListHandler listhandler(datasetFileName);
00060     auto start = std::chrono::high_resolution_clock::now();
00061
00062     std::vector<std::thread*> threadsV;
00063     for(int i=0;i<threads;i++){
00064         threadsV.push_back(new std::thread(&Markov::API::MarkovPasswords::TrainThread, this,
       &listhandler, delimiter));
00065     }
00066
00067     for(int i=0;i<threads;i++){
00068         threadsV[i]->join();
00069         delete threadsV[i];
00070     }
00071     auto finish = std::chrono::high_resolution_clock::now();
00072     std::chrono::duration<double> elapsed = finish - start;
00073     std::cout « "Elapsed time: " « elapsed.count() « " s\n";
00074
00075 }
00076
00077 void Markov::API::MarkovPasswords::TrainThread(Markov::API::Concurrency::ThreadSharedListHandler
       *listhandler, char delimiter){
00078     char format_str[] ="%ld,%s";
00079     format_str[2]=delimiter;
00080     std::string line;
00081     while (listhandler->next(&line) && keepRunning) {
00082         long int oc;
00083         if (line.size() > 100) {
```

```
00084             line = line.substr(0, 100);
00085         }
00086         char* linebuf = new char[line.length()+5];
00087 #ifdef _WIN32
00088         sscanf_s(line.c_str(), "%ld,%s", &oc, linebuf, line.length()+5); //<== changed format_str to->
       "%ld,%s"
00089 #else
00090         sscanf(line.c_str(), format_str, &oc, linebuf);
00091 #endif
00092         this->AdjustEdge((const char*)linebuf, oc);
00093         delete linebuf;
00094     }
00095 }
00096
00097
00098 std::ofstream* Markov::API::MarkovPasswords::Save(const char* filename) {
00099     std::ofstream* exportFile;
00100
00101     std::ofstream newFile(filename);
00102
00103     exportFile = &newFile;
00104
00105     this->Export(exportFile);
00106     return exportFile;
00107 }
00108
00109
00110 void Markov::API::MarkovPasswords::Generate(unsigned long int n, const char* wordlistFileName, int
       minLen, int maxLen, int threads)  {
00111     char* res;
00112     char print[100];
00113     std::ofstream wordlist;
00114     wordlist.open(wordlistFileName);
00115     std::mutex mlock;
00116     int iterationsPerThread = n/threads;
00117     int iterationsCarryOver = n%threads;
00118     std::vector<std::thread*> threadsV;
00119     for(int i=0;i<threads;i++){
00120         threadsV.push_back(new std::thread(&Markov::API::MarkovPasswords::GenerateThread, this,
       &mlock, iterationsPerThread, &wordlist, minLen, maxLen));
00121     }
00122
00123     for(int i=0;i<threads;i++){
00124         threadsV[i]->join();
00125         delete threadsV[i];
00126     }
00127
00128     this->GenerateThread(&mlock, iterationsCarryOver, &wordlist, minLen, maxLen);
00129
00130 }
00131
00132 void Markov::API::MarkovPasswords::GenerateThread(std::mutex *outputLock, unsigned long int n,
       std::ofstream *wordlist, int minLen, int maxLen)  {
00133     char* res = new char[maxLen+5];
00134     if(n==0) return;
00135
00136     Markov::Random::Marsaglia MarsagliaRandomEngine;
00137     for (int i = 0; i < n; i++) {
00138         this->RandomWalk(&MarsagliaRandomEngine, minLen, maxLen, res);
00139         outputLock->lock();
00140         *wordlist « res « "\n";
00141         outputLock->unlock();
00142     }
00143 }
```

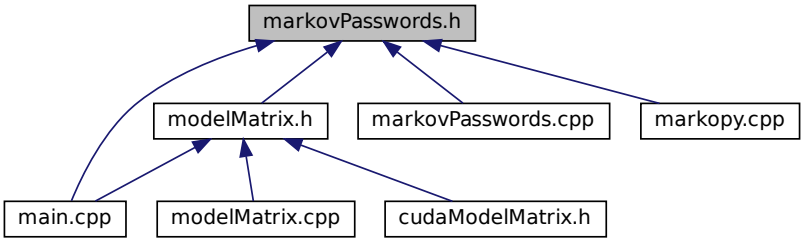
9.31 markovPasswords.h File Reference

```
#include "threadSharedListHandler.h"
#include "MarkovModel/src/model.h"
```

Include dependency graph for markovPasswords.h:



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



Classes

- class Markov::API::MarkovPasswords

  Markov::Model with char represented nodes.

Namespaces

- Markov

  Namespace for the markov-model related classes. Contains Model, Node and Edge classes.
- Markov::API

  Namespace for the MarkovPasswords API.

9.32 markovPasswords.h

```
00001 #pragma once
00002 #include "threadSharedListHandler.h"
00003 #include "MarkovModel/src/model.h"
00004
00005
00006 /** @brief Namespace for the MarkovPasswords API
00007 */
00008 namespace Markov::API{
00009
00010     /** @brief Markov::Model with char represented nodes.
00011      *
00012      * Includes wrappers for Markov::Model and additional helper functions to handle file I/O
00013      *
```

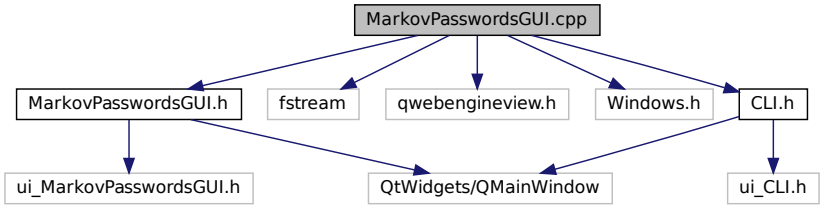
```
00014     * This class is an extension of Markov::Model<char>, with higher level abstractions such as train
      and generate.
00015     *
00016     */
00017    class MarkovPasswords : public Markov::Model<char>{
00018    public:
00019
00020        /** @brief Initialize the markov model from MarkovModel::Markov::Model.
00021         *
00022         * Parent constructor. Has no extra functionality.
00023         */
00024        MarkovPasswords();
00025
00026        /** @brief Initialize the markov model from MarkovModel::Markov::Model, with an import file.
00027         *
00028         * This function calls the Markov::Model::Import on the filename to construct the model.
00029         * Same thing as creating and empty model, and calling MarkovPasswords::Import on the
      filename.
00030         *
00031         * @param filename - Filename to import
00032         *
00033         *
00034         * @b Example @b Use: Construction via filename
00035         * @code{.cpp}
00036         * MarkovPasswords mp("test.mdl");
00037         * @endcode
00038         */
00039        MarkovPasswords(const char* filename);
00040
00041        /** @brief Open dataset file and return the ifstream pointer
00042         * @param filename - Filename to open
00043         * @return ifstream* to the the dataset file
00044         */
00045        std::ifstream* OpenDatasetFile(const char* filename);
00046
00047
00048        /** @brief Train the model with the dataset file.
00049         * @param datasetFileName - Ifstream* to the dataset. If null, use class member
00050         * @param delimiter - a character, same as the delimiter in dataset content
00051         * @param threads - number of OS threads to spawn
00052         *
00053         * @code{.cpp}
00054         * Markov::API::MarkovPasswords mp;
00055         * mp.Import("models/2gram.mdl");
00056         * mp.Train("password.corpus");
00057         * @endcode
00058         */
00059        void Train(const char* datasetFileName, char delimiter, int threads);
00060
00061
00062
00063        /** @brief Export model to file.
00064        * @param filename - Export filename.
00065        * @return std::ofstream* of the exported file.
00066        */
00067        std::ofstream* Save(const char* filename);
00068
00069        /** @brief Call Markov::Model::RandomWalk n times, and collect output.
00070         *
00071         * Generate from model and write results to a file.
00072         * a much more performance-optimized method. FastRandomWalk will reduce the runtime by %96.5
      on average.
00073         *
00074         * @deprecated See Markov::API::MatrixModel::FastRandomWalk for more information.
00075         * @param n - Number of passwords to generate.
00076         * @param wordlistFileName - Filename to write to
00077         * @param minLen - Minimum password length to generate
00078         * @param maxLen - Maximum password length to generate
00079         * @param threads - number of OS threads to spawn
00080         */
00081        void Generate(unsigned long int n, const char* wordlistFileName, int minLen=6, int maxLen=12,
      int threads=20);
00082
00083
00084    private:
00085
00086        /** @brief A single thread invoked by the Train function.
00087         * @param listhandler - Listhandler class to read corpus from
00088         * @param delimiter - a character, same as the delimiter in dataset content
00089         *
00090         */
00091        void TrainThread(Markov::API::Concurrency::ThreadSharedListHandler *listhandler, char
      delimiter);
00092
00093        /** @brief A single thread invoked by the Generate function.
00094         *
00095         * @b DEPRECATED: See Markov::API::MatrixModel::FastRandomWalkThread for more information.
```

```
      This has been replaced with
00096         * a much more performance-optimized method. FastRandomWalk will reduce the runtime by %96.5
      on average.
00097         *
00098         * @param outputLock - shared mutex lock to lock during output operation. Prevents race
      condition on write.
00099         * @param n number of lines to be generated by this thread
00100         * @param wordlist wordlistfile
00101         * @param minLen - Minimum password length to generate
00102         * @param maxLen - Maximum password length to generate
00103         *
00104         */
00105        void GenerateThread(std::mutex *outputLock, unsigned long int n, std::ofstream *wordlist, int
      minLen, int maxLen);
00106        std::ifstream* datasetFile; /** @brief  Dataset file input of our system    */
00107        std::ofstream* modelSavefile; /** @brief    File to save model  of our system   */
00108        std::ofstream* outputFile; /** @brief   Generated output  file  of our system   */
00109    };
00110
00111
00112
00113 };
```

9.33 MarkovPasswordsGUI.cpp File Reference

```
#include "MarkovPasswordsGUI.h"
#include <fstream>
#include <qwebengineview.h>
#include <Windows.h>
#include "CLI.h"
```

Include dependency graph for MarkovPasswordsGUI.cpp:



9.34 MarkovPasswordsGUI.cpp

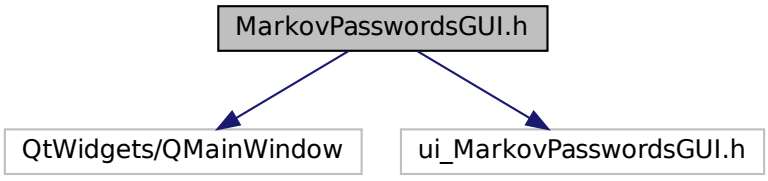
```
00001 #include "MarkovPasswordsGUI.h"
00002 #include <fstream>
00003 #include <qwebengineview.h>
00004 #include <Windows.h>
00005 #include "CLI.h"
00006
00007 using namespace Markov::GUI;
00008
00009 MarkovPasswordsGUI::MarkovPasswordsGUI(QWidget *parent)
00010     : QMainWindow(parent)
00011 {
00012     ui.setupUi(this);
00013
00014
00015     QObject::connect(ui.pushButton, &QPushButton::clicked, this, [this] {home(); });
00016     QObject::connect(ui.pushButton_2, &QPushButton::clicked, this, [this] {model(); });
00017     QObject::connect(ui.pushButton_3, &QPushButton::clicked, this, [this] {pass(); });
00018 }
00019
00020
00021 void MarkovPasswordsGUI::home() {
00022     CLI* w = new CLI;
00023     w->show();
00024     this->close();
00025 }
00026 void MarkovPasswordsGUI::pass() {
00027     QWebEngineView* webkit = ui.centralWidget->findChild<QWebEngineView*>("chartArea");
```

```
00028
00029     //get working directory
00030     char path[255];
00031     GetCurrentDirectoryA(255, path);
00032
00033     //get absolute path to the layout html
00034     std::string layout = "file:///" + std::string(path) + "\\views\\bar.html";
00035     std::replace(layout.begin(), layout.end(), '\\', '/');
00036     webkit->setUrl(QUrl(layout.c_str()));
00037 }
00038
00039 void MarkovPasswordsGUI::model() {
00040     QWebEngineView* webkit = ui.centralWidget->findChild<QWebEngineView*>("chartArea");
00041
00042     //get working directory
00043     char path[255];
00044     GetCurrentDirectoryA(255, path);
00045
00046     //get absolute path to the layout html
00047     std::string layout = "file:///" + std::string(path) + "\\views\\index.html";
00048     std::replace(layout.begin(), layout.end(), '\\', '/');
00049     webkit->setUrl(QUrl(layout.c_str()));
00050 }
```

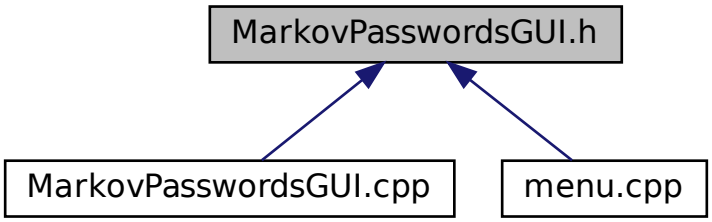
9.35 MarkovPasswordsGUI.h File Reference

```
#include <QtWidgets/QMainWindow>
#include "ui_MarkovPasswordsGUI.h"
```

Include dependency graph for MarkovPasswordsGUI.h:



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



Classes

- class Markov::GUI::MarkovPasswordsGUI

  Reporting UI.

Namespaces

- Markov

  Namespace for the markov-model related classes. Contains Model, Node and Edge classes.
- Markov::GUI

  namespace for MarkovPasswords API GUI wrapper

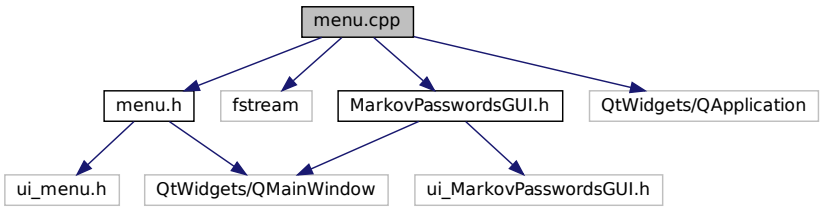
9.36 MarkovPasswordsGUI.h

```
00001 #pragma once
00002
00003 #include <QtWidgets/QMainWindow>
00004 #include "ui_MarkovPasswordsGUI.h"
00005
00006
00007
00008 namespace Markov::GUI{
00009     /** @brief Reporting UI.
00010     *
00011     * UI for reporting and debugging tools for MarkovPassword
00012     */
00013     class MarkovPasswordsGUI : public QMainWindow {
00014         Q_OBJECT
00015
00016     private:
00017         Ui::MarkovPasswordsGUIClass ui;
00018
00019
00020         //Slots for buttons in GUI.
00021     public slots:
00022
00023         void MarkovPasswordsGUI::benchmarkSelected();
00024         void MarkovPasswordsGUI::modelvisSelected();
00025         void MarkovPasswordsGUI::visualDebugSelected();
00026         void MarkovPasswordsGUI::comparisonSelected();
00027
00028
00029   public slots:
00030
00031       void MarkovPasswordsGUI::home();
00032       void MarkovPasswordsGUI :: pass();
00033       void MarkovPasswordsGUI::model();
00034   };
00035 };
```

9.37 menu.cpp File Reference

```
#include "menu.h"
#include <fstream>
#include "MarkovPasswordsGUI.h"
#include <QtWidgets/QApplication>
```

Include dependency graph for menu.cpp:



9.38 menu.cpp

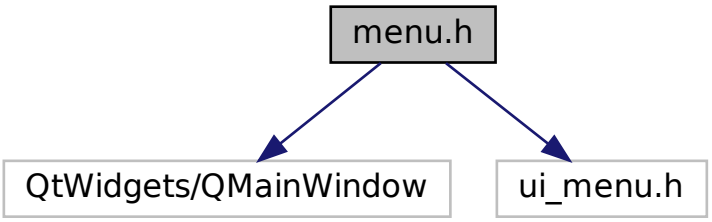
```
00001 #include "menu.h"
```

```
00002 #include <fstream>
00003 #include "MarkovPasswordsGUI.h"
00004 #include <QtWidgets/QApplication>
00005
00006 using namespace Markov::GUI;
00007
00008 menu::menu(QWidget* parent)
00009     : QMainWindow(parent)
00010 {
00011     ui.setupUi(this);
00012
00013
00014     //QObject::connect(ui.pushButton, &QPushButton::clicked, this, [this] {about(); });
00015     QObject::connect(ui.visu, &QPushButton::clicked, this, [this] {visualization(); });
00016 }
00017 void menu::about() {
00018
00019
00020 }
00021 void menu::visualization() {
00022     MarkovPasswordsGUI* w = new MarkovPasswordsGUI;
00023     w->show();
00024     this->close();
00025 }
```

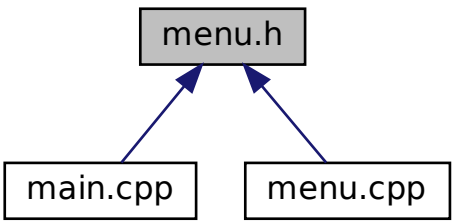
9.39 menu.h File Reference

```
#include <QtWidgets/QMainWindow>
#include "ui_menu.h"
```

Include dependency graph for menu.h:



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



Classes

- class Markov::GUI::menu

  QT Menu class.

Namespaces

- Markov

  Namespace for the markov-model related classes. Contains Model, Node and Edge classes.
- Markov::GUI

  namespace for MarkovPasswords API GUI wrapper

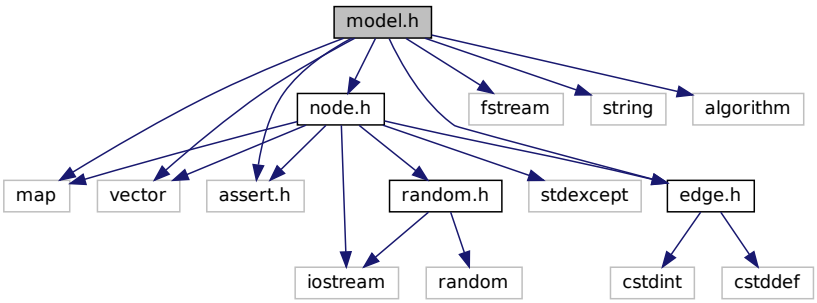
9.40 menu.h

```
00001 #pragma once
00002 #include <QtWidgets/QMainWindow>
00003 #include "ui_menu.h"
00004
00005
00006 namespace Markov::GUI{
00007     /** @brief QT Menu class
00008     */
00009     class menu:public QMainWindow {
00010     Q_OBJECT
00011     public:
00012         menu(QWidget* parent = Q_NULLPTR);
00013
00014     private:
00015         Ui::main ui;
00016
00017     public slots:
00018         void about();
00019         void visualization();
00020     };
00021 };
```

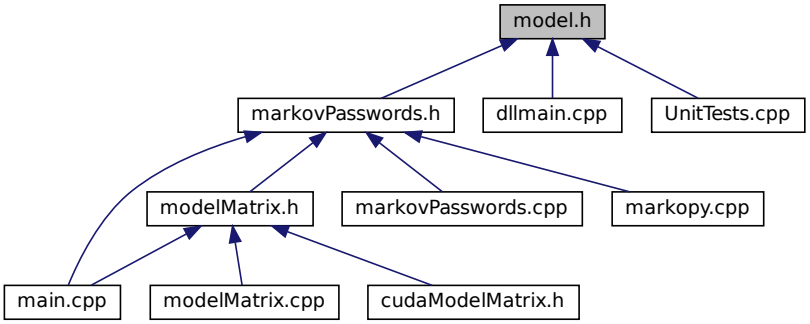
9.41 model.h File Reference

```
#include <map>
#include <vector>
#include <fstream>
#include <assert.h>
#include <string>
#include <algorithm>
#include "node.h"
#include "edge.h"
```

Include dependency graph for model.h:



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



Classes

- class Markov::Node< storageType >

  A node class that for the vertices of model. Connected with eachother using Edge.
- class Markov::Edge< NodeStorageType >

  Edge class used to link nodes in the model together.
- class Markov::Model< NodeStorageType >

  class for the final Markov Model, constructed from nodes and edges.

Namespaces

- Markov

  Namespace for the markov-model related classes. Contains Model, Node and Edge classes.

9.42 model.h

```
00001 /** @dir Model.h
00002  *
00003  */
00004
00005
00006 #pragma once
00007 #include <map>
00008 #include <vector>
00009 #include <fstream>
00010 #include <assert.h>
00011 #include <string>
00012 #include <algorithm>
00013 #include "node.h"
00014 #include "edge.h"
00015
00016 /**
00017     @brief Namespace for the markov-model related classes.
00018     Contains Model, Node and Edge classes
00019 */
00020 namespace Markov {
00021
00022     template <typename NodeStorageType>
00023     class Node;
00024
00025     template <typename NodeStorageType>
00026     class Edge;
00027
00028     template <typename NodeStorageType>
00029
00030     /** @brief class for the final Markov Model, constructed from nodes and edges.
00031      *
00032      * Each atomic piece of the generation result is stored in a node, while edges contain the
       relation weights.
00033      * *Extending:*
```

```
00034     * To extend the class, implement the template and inherit from it, as "class MyModel : public
      Markov::Model<char>".
00035     * For a complete demonstration of how to extend the class, see MarkovPasswords.
00036     *
00037     * Whole model can be defined as a list of the edges, as dangling nodes are pointless. This
      approach is used for the import/export operations.
00038     * For more information on importing/exporting model, check out the github readme and wiki page.
00039     *
00040    */
00041    class Model {
00042    public:
00043
00044        /** @brief Initialize a model with only start and end nodes.
00045         *
00046         * Initialize an empty model with only a starterNode
00047         * Starter node is a special kind of node that has constant 0x00 value, and will be used to
      initiate the generation execution from.
00048        */
00049        Model<NodeStorageType>();
00050
00051        /** @brief Do a random walk on this model.
00052         *
00053         * Start from the starter node, on each node, invoke RandomNext using the random engine on
      current node, until terminator node is reached.
00054         * If terminator node is reached before minimum length criateria is reached, ignore the last
      selection and re-invoke randomNext
00055         *
00056         * If maximum length criteria is reached but final node is not, cut off the generation and
      proceed to the final node.
00057         * This function takes Markov::Random::RandomEngine as a parameter to generate pseudo random
      numbers from
00058         *
00059         * This library is shipped with two random engines, Marsaglia and Mersenne. While mersenne
      output is higher in entropy, most use cases
00060         * don't really need super high entropy output, so Markov::Random::Marsaglia is preferable for
      better performance.
00061         *
00062         * This function WILL NOT reallocate buffer. Make sure no out of bound writes are happening
      via maximum length criteria.
00063         *
00064         * @b Example @b Use: Generate 10 lines, with 5 to 10 characters, and print the output. Use
      Marsaglia
00065         * @code{.cpp}
00066         * Markov::Model<char> model;
00067         * Model.import("model.mdl");
00068         * char* res = new char[11];
00069         * Markov::Random::Marsaglia MarsagliaRandomEngine;
00070         * for (int i = 0; i < 10; i++) {
00071         *      this->RandomWalk(&MarsagliaRandomEngine, 5, 10, res);
00072         *      std::cout « res « "\n";
00073         *  }
00074         * @endcode
00075         *
00076         * @param randomEngine Random Engine to use for the random walks. For examples, see
      Markov::Random::Mersenne and Markov::Random::Marsaglia
00077         * @param minSetting Minimum number of characters to generate
00078         * @param maxSetting Maximum number of character to generate
00079         * @param buffer buffer to write the result to
00080         * @return Null terminated string that was generated.
00081        */
00082        NodeStorageType* RandomWalk(Markov::Random::RandomEngine* randomEngine, int minSetting, int
      maxSetting, NodeStorageType* buffer);
00083
00084        /** @brief Adjust the model with a single string.
00085         *
00086         * Start from the starter node, and for each character, AdjustEdge the edge EdgeWeight from
      current node to the next, until NULL character is reached.
00087         *
00088         * Then, update the edge EdgeWeight from current node, to the terminator node.
00089         *
00090         * This function is used for training purposes, as it can be used for adjusting the model with
      each line of the corpus file.
00091         *
00092         * @b Example @b Use: Create an empty model and train it with string: "testdata"
00093         * @code{.cpp}
00094         * Markov::Model<char> model;
00095         * char test[] = "testdata";
00096         * model.AdjustEdge(test, 15);
00097         * @endcode
00098         *
00099         *
00100         * @param string - String that is passed from the training, and will be used to AdjustEdge the
      model with
00101         * @param occurrence - Occurrence of this string.
00102         *
00103         *
00104        */
```

```
00105         void AdjustEdge(const NodeStorageType* payload, long int occurrence);
00106
00107         /** @brief Import a file to construct the model.
00108          *
00109          * File contains a list of edges. For more info on the file format, check out the wiki and
      github readme pages.
00110          * Format is: Left_repr;EdgeWeight;right_repr
00111          *
00112          * Iterate over this list, and construct nodes and edges accordingly.
00113          * @return True if successful, False for incomplete models or corrupt file formats
00114          *
00115          * @b Example @b Use: Import a file from ifstream
00116          * @code{.cpp}
00117          * Markov::Model<char> model;
00118          * std::ifstream file("test.mdl");
00119          * model.Import(&file);
00120          * @endcode
00121         */
00122         bool Import(std::ifstream*);
00123
00124         /** @brief Open a file to import with filename, and call bool Model::Import with std::ifstream
00125          * @return True if successful, False for incomplete models or corrupt file formats
00126          *
00127          * @b Example @b Use: Import a file with filename
00128          * @code{.cpp}
00129          * Markov::Model<char> model;
00130          * model.Import("test.mdl");
00131          * @endcode
00132         */
00133         bool Import(const char* filename);
00134
00135         /** @brief Export a file of the model.
00136          *
00137          * File contains a list of edges.
00138          * Format is: Left_repr;EdgeWeight;right_repr.
00139          * For more information on the format, check out the project wiki or github readme.
00140          *
00141          * Iterate over this vertices, and their edges, and write them to file.
00142          * @return True if successful, False for incomplete models.
00143          *
00144          * @b Example @b Use: Export file to ofstream
00145          * @code{.cpp}
00146          * Markov::Model<char> model;
00147          * std::ofstream file("test.mdl");
00148          * model.Export(&file);
00149          * @endcode
00150         */
00151         bool Export(std::ofstream*);
00152
00153         /** @brief Open a file to export with filename, and call bool Model::Export with std::ofstream
00154          * @return True if successful, False for incomplete models or corrupt file formats
00155          *
00156          * @b Example @b Use: Export file to filename
00157          * @code{.cpp}
00158          * Markov::Model<char> model;
00159          * model.Export("test.mdl");
00160          * @endcode
00161         */
00162         bool Export(const char* filename);
00163
00164         /** @brief Return starter Node
00165          * @return starter node with 00 NodeValue
00166         */
00167         Node<NodeStorageType>* StarterNode(){ return starterNode;}
00168
00169         /** @brief Return a vector of all the edges in the model
00170          * @return vector of edges
00171         */
00172         std::vector<Edge<NodeStorageType>*>* Edges(){ return &edges;}
00173
00174         /** @brief Return starter Node
00175          * @return starter node with 00 NodeValue
00176         */
00177         std::map<NodeStorageType, Node<NodeStorageType>*>* Nodes(){ return &nodes;}
00178
00179     private:
00180         /**
00181             @brief Map LeftNode is the Nodes NodeValue
00182             * Map RightNode is the node pointer
00183         */
00184         std::map<NodeStorageType, Node<NodeStorageType>*> nodes;
00185
00186         /**
00187             @brief Starter Node of this model.
00188         */
00189         Node<NodeStorageType>* starterNode;
00190
```

```
00191
00192         /**
00193             @brief A list of all edges in this model.
00194         */
00195         std::vector<Edge<NodeStorageType>*> edges;
00196     };
00197
00198 };
00199
00200 template <typename NodeStorageType>
00201 Markov::Model<NodeStorageType>::Model() {
00202     this->starterNode = new Markov::Node<NodeStorageType>(0);
00203     this->nodes.insert({ 0, this->starterNode });
00204 }
00205
00206 template <typename NodeStorageType>
00207 bool Markov::Model<NodeStorageType>::Import(std::ifstream* f) {
00208     std::string cell;
00209
00210     char src;
00211     char target;
00212     long int oc;
00213
00214     while (std::getline(*f, cell)) {
00215         //std::cout « "cell: " « cell « std::endl;
00216         src = cell[0];
00217         target = cell[cell.length() - 1];
00218         char* j;
00219         oc = std::strtol(cell.substr(2, cell.length() - 2).c_str(),&j,10);
00220         //std::cout « oc « "\n";
00221         Markov::Node<NodeStorageType>* srcN;
00222         Markov::Node<NodeStorageType>* targetN;
00223         Markov::Edge<NodeStorageType>* e;
00224         if (this->nodes.find(src) == this->nodes.end()) {
00225             srcN = new Markov::Node<NodeStorageType>(src);
00226             this->nodes.insert(std::pair<char, Markov::Node<NodeStorageType>*>(src, srcN));
00227             //std::cout « "Creating new node at start.\n";
00228         }
00229         else {
00230             srcN = this->nodes.find(src)->second;
00231         }
00232
00233         if (this->nodes.find(target) == this->nodes.end()) {
00234             targetN = new Markov::Node<NodeStorageType>(target);
00235             this->nodes.insert(std::pair<char, Markov::Node<NodeStorageType>*>(target, targetN));
00236             //std::cout « "Creating new node at end.\n";
00237         }
00238         else {
00239             targetN = this->nodes.find(target)->second;
00240         }
00241         e = srcN->Link(targetN);
00242         e->AdjustEdge(oc);
00243         this->edges.push_back(e);
00244
00245         //std::cout « int(srcN->NodeValue()) « " --" « e->EdgeWeight() « "--> " «
      int(targetN->NodeValue()) « "\n";
00246
00247
00248     }
00249
00250     for (std::pair<unsigned char, Markov::Node<NodeStorageType>*> const& x : this->nodes) {
00251         //std::cout « "Total edges in EdgesV: " « x.second->edgesV.size() « "\n";
00252         std::sort (x.second->edgesV.begin(), x.second->edgesV.end(), [](Edge<NodeStorageType> *lhs,
      Edge<NodeStorageType> *rhs)->bool{
00253             return lhs->EdgeWeight() > rhs->EdgeWeight();
00254         });
00255         //for(int i=0;i<x.second->edgesV.size();i++)
00256         //  std::cout « x.second->edgesV[i]->EdgeWeight() « ", ";
00257         //std::cout « "\n";
00258     }
00259     //std::cout « "Total number of nodes: " « this->nodes.size() « std::endl;
00260     //std::cout « "Total number of edges: " « this->edges.size() « std::endl;
00261
00262     return true;
00263 }
00264
00265 template <typename NodeStorageType>
00266 bool Markov::Model<NodeStorageType>::Import(const char* filename) {
00267     std::ifstream importfile;
00268     importfile.open(filename);
00269     return this->Import(&importfile);
00270
00271 }
00272
00273 template <typename NodeStorageType>
00274 bool Markov::Model<NodeStorageType>::Export(std::ofstream* f) {
00275     Markov::Edge<NodeStorageType>* e;
```

```
00276     for (std::vector<int>::size_type i = 0; i != this->edges.size(); i++) {
00277         e = this->edges[i];
00278         //std::cout << e->LeftNode()->NodeValue() << "," << e->EdgeWeight() << "," <<
      e->RightNode()->NodeValue() << "\n";
00279         *f << e->LeftNode()->NodeValue() << "," << e->EdgeWeight() << "," << e->RightNode()->NodeValue() <<
      "\n";
00280     }
00281
00282     return true;
00283 }
00284
00285 template <typename NodeStorageType>
00286 bool Markov::Model<NodeStorageType>::Export(const char* filename) {
00287     std::ofstream exportfile;
00288     exportfile.open(filename);
00289     return this->Export(&exportfile);
00290 }
00291
00292 template <typename NodeStorageType>
00293 NodeStorageType* Markov::Model<NodeStorageType>::RandomWalk(Markov::Random::RandomEngine*
      randomEngine, int minSetting, int maxSetting, NodeStorageType* buffer) {
00294     Markov::Node<NodeStorageType>* n = this->starterNode;
00295     int len = 0;
00296     Markov::Node<NodeStorageType>* temp_node;
00297     while (true) {
00298         temp_node = n->RandomNext(randomEngine);
00299         if (len >= maxSetting) {
00300             break;
00301         }
00302         else if ((temp_node == NULL) && (len < minSetting)) {
00303             continue;
00304         }
00305
00306         else if (temp_node == NULL){
00307             break;
00308         }
00309
00310         n = temp_node;
00311
00312         buffer[len++] = n->NodeValue();
00313     }
00314
00315     //null terminate the string
00316     buffer[len] = 0x00;
00317
00318     //do something with the generated string
00319     return buffer; //for now
00320 }
00321
00322 template <typename NodeStorageType>
00323 void Markov::Model<NodeStorageType>::AdjustEdge(const NodeStorageType* payload, long int occurrence) {
00324     NodeStorageType p = payload[0];
00325     Markov::Node<NodeStorageType>* curnode = this->starterNode;
00326     Markov::Edge<NodeStorageType>* e;
00327     int i = 0;
00328
00329     if (p == 0) return;
00330     while (p != 0) {
00331         e = curnode->FindEdge(p);
00332         if (e == NULL) return;
00333         e->AdjustEdge(occurrence);
00334         curnode = e->RightNode();
00335         p = payload[++i];
00336     }
00337
00338     e = curnode->FindEdge('\xff');
00339     e->AdjustEdge(occurrence);
00340     return;
00341 }
```

9.43 model_2gram.py File Reference

Namespaces

- model_2gram

Variables

- model_2gram.alphabet = string.printable

  password alphabet

- model_2gram.f = open('../../models/2gram.mdl', "wb")

  output file handle

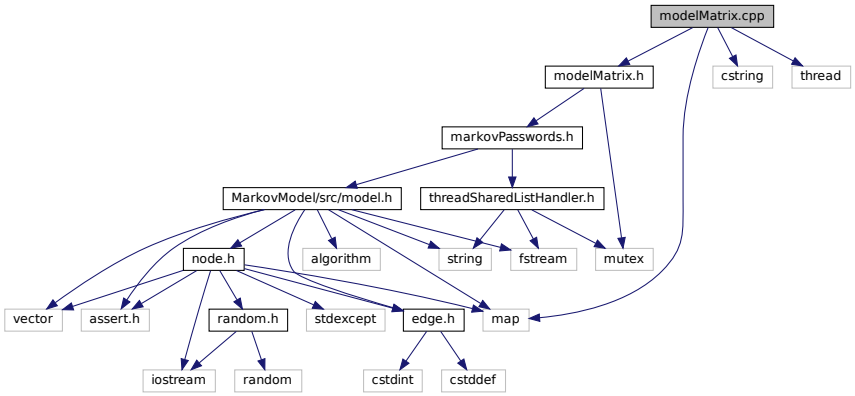
9.44 model_2gram.py

```
00001 #!/usr/bin/python3
00002 """
00003   python script for generating a 2gram model
00004 """
00005
00006 import string
00007 import re
00008
00009
00010 alphabet = string.printable
00011 alphabet = re.sub('\s', '', alphabet)
00012 print(f"alphabet={alphabet}")
00013 #exit()
00014
00015
00016 f = open('../../models/2gram.mdl', "wb")
00017 #tie start nodes
00018 for sym in alphabet:
00019     f.write(b"\x00,1," + bytes(sym, encoding='ascii') + b"\n")
00020
00021 #tie terminator nodes
00022 for sym in alphabet:
00023     f.write(bytes(sym, encoding='ascii')+ b",1,\xff\n")
00024
00025 #tie internals
00026 for src in alphabet:
00027     for target in alphabet:
00028         f.write(bytes(src, encoding='ascii') + b",1," + bytes(target, encoding='ascii') + b"\n")
```

9.45 modelMatrix.cpp File Reference

```
#include "modelMatrix.h"
#include <map>
#include <cstring>
#include <thread>
```

Include dependency graph for modelMatrix.cpp:



9.46 modelMatrix.cpp

```
00001 #include "modelMatrix.h"
00002 #include <map>
00003 #include <cstring>
00004 #include <thread>
00005
00006 Markov::API::ModelMatrix::ModelMatrix(){
00007
```

```
00008 }
00009
00010
00011 void Markov::API::ModelMatrix::ConstructMatrix(){
00012     this->matrixSize = this->StarterNode()->edgesV.size() + 2;
00013
00014     this->matrixIndex = new char[this->matrixSize];
00015     this->totalEdgeWeights = new long int[this->matrixSize];
00016
00017     this->edgeMatrix = new char*[this->matrixSize];
00018     for(int i=0;i<this->matrixSize;i++){
00019         this->edgeMatrix[i] = new char[this->matrixSize];
00020     }
00021     this->valueMatrix = new long int*[this->matrixSize];
00022     for(int i=0;i<this->matrixSize;i++){
00023         this->valueMatrix[i] = new long int[this->matrixSize];
00024     }
00025     std::map< char, Node< char > * > *nodes;
00026     nodes = this->Nodes();
00027     int i=0;
00028     for (auto const& [repr, node] : *nodes){
00029         if(repr!=0) this->matrixIndex[i] = repr;
00030         else this->matrixIndex[i] = 199;
00031         this->totalEdgeWeights[i] = node->TotalEdgeWeights();
00032         for(int j=0;j<this->matrixSize;j++){
00033             char val = node->NodeValue();
00034             if(val < 0){
00035                 for(int k=0;k<this->matrixSize;k++){
00036                     this->valueMatrix[i][k] = 0;
00037                     this->edgeMatrix[i][k] = 255;
00038                 }
00039                 break;
00040             }
00041             else if(node->NodeValue() == 0 && j>(this->matrixSize-3)){
00042                 this->valueMatrix[i][j] = 0;
00043                 this->edgeMatrix[i][j] = 255;
00044             }else if(j==(this->matrixSize-1)) {
00045                 this->valueMatrix[i][j] = 0;
00046                 this->edgeMatrix[i][j] = 255;
00047             }else{
00048                 this->valueMatrix[i][j] = node->edgesV[j]->EdgeWeight();
00049                 this->edgeMatrix[i][j]  = node->edgesV[j]->RightNode()->NodeValue();
00050             }
00051
00052         }
00053         i++;
00054     }
00055
00056     //this->DumpJSON();
00057 }
00058
00059
00060 void Markov::API::ModelMatrix::DumpJSON(){
00061
00062     std::cout << "{\n   \"index\": \"";
00063     for(int i=0;i<this->matrixSize;i++){
00064         if(this->matrixIndex[i]=='"') std::cout << "\\\"";
00065         else if(this->matrixIndex[i]=='\\') std::cout << "\\\\";
00066         else if(this->matrixIndex[i]==0) std::cout << "\\\\x00";
00067         else if(i==0) std::cout << "\\\\xff";
00068         else if(this->matrixIndex[i]=='\n') std::cout << "\\n";
00069         else std::cout << this->matrixIndex[i];
00070     }
00071     std::cout <<
00072     "\",\n"
00073     "   \"edgemap\": {\n";
00074
00075     for(int i=0;i<this->matrixSize;i++){
00076         if(this->matrixIndex[i]=='"') std::cout << "      \"\\\"\": [";
00077         else if(this->matrixIndex[i]=='\\') std::cout << "      \"\\\\\": [";
00078         else if(this->matrixIndex[i]==0) std::cout << "      \"\\\\x00\": [";
00079         else if(this->matrixIndex[i]<0) std::cout << "      \"\\\\xff\": [";
00080         else std::cout << "      \"" << this->matrixIndex[i] << "\": [";
00081         for(int j=0;j<this->matrixSize;j++){
00082             if(this->edgeMatrix[i][j]=='"') std::cout << "\"\\\"\"";
00083             else if(this->edgeMatrix[i][j]=='\\') std::cout << "\"\\\\\"";
00084             else if(this->edgeMatrix[i][j]==0) std::cout << "\"\\\\x00\"";
00085             else if(this->edgeMatrix[i][j]<0) std::cout << "\"\\\\xff\"";
00086             else if(this->matrixIndex[i]=='\n') std::cout << "\"\\n\"";
00087             else std::cout << "\"" << this->edgeMatrix[i][j] << "\"";
00088             if(j!=this->matrixSize-1) std::cout << ", ";
00089         }
00090         std::cout << "],\n";
00091     }
00092     std::cout << "},\n";
00093
00094     std::cout << "\"   weightmap\": {\n";
```

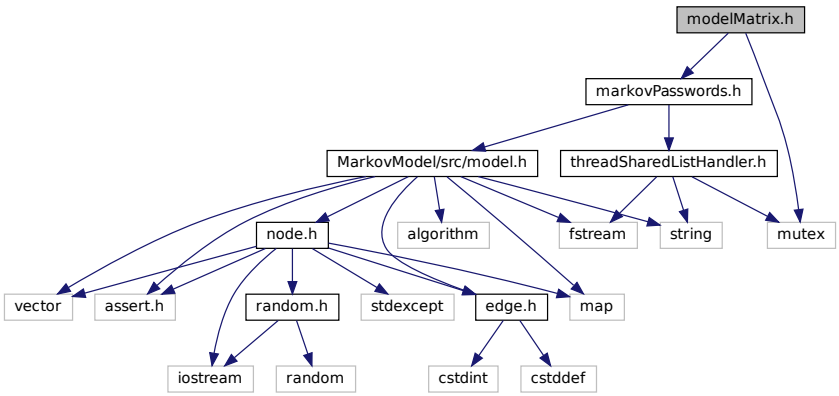
```
00095     for(int i=0;i<this->matrixSize;i++){
00096         if(this->matrixIndex[i]=='"') std::cout << "      \"\\\"\": [";
00097         else if(this->matrixIndex[i]=='\\') std::cout << "      \"\\\\\": [";
00098         else if(this->matrixIndex[i]==0) std::cout << "      \"\\\\x00\": [";
00099         else if(this->matrixIndex[i]<0) std::cout << "      \"\\\\xff\": [";
00100         else std::cout << "      \"" << this->matrixIndex[i] << "\": [";
00101 
00102         for(int j=0;j<this->matrixSize;j++){
00103             std::cout << this->valueMatrix[i][j];
00104             if(j!=this->matrixSize-1) std::cout << ", ";
00105         }
00106         std::cout << "],\n";
00107     }
00108     std::cout << "  }\n}\n";
00109 }
00110 
00111 
00112 void Markov::API::ModelMatrix::FastRandomWalkThread(std::mutex *mlock, std::ofstream *wordlist,
      unsigned long int n, int minLen, int maxLen, int id, bool bFileIO){
00113     if(n==0) return;
00114 
00115     Markov::Random::Marsaglia MarsagliaRandomEngine;
00116     char* e;
00117     char *res = new char[maxLen*n];
00118     int index = 0;
00119     char next;
00120     int len=0;
00121     long int selection;
00122     char cur;
00123     long int bufferctr = 0;
00124     for (int i = 0; i < n; i++) {
00125         cur=199;
00126         len=0;
00127         while (true) {
00128             e = strchr(this->matrixIndex, cur);
00129             index = e - this->matrixIndex;
00130             selection = MarsagliaRandomEngine.random() % this->totalEdgeWeights[index];
00131             for(int j=0;j<this->matrixSize;j++){
00132                 selection -= this->valueMatrix[index][j];
00133                 if (selection < 0){
00134                     next = this->edgeMatrix[index][j];
00135                     break;
00136                 }
00137             }
00138 
00139             if (len >= maxLen)  break;
00140             else if ((next < 0) && (len < minLen)) continue;
00141             else if (next < 0) break;
00142             cur = next;
00143             res[bufferctr + len++] = cur;
00144         }
00145         res[bufferctr + len++] = '\n';
00146         bufferctr+=len;
00147 
00148     }
00149     if(bFileIO){
00150         mlock->lock();
00151         *wordlist << res;
00152         mlock->unlock();
00153     }else{
00154         mlock->lock();
00155         std::cout << res;
00156         mlock->unlock();
00157     }
00158     delete res;
00159 
00160 }
00161 
00162 
00163 void Markov::API::ModelMatrix::FastRandomWalk(unsigned long int n, const char* wordlistFileName, int
      minLen, int maxLen, int threads, bool bFileIO){
00164 
00165 
00166     std::ofstream wordlist;
00167     if(bFileIO)
00168         wordlist.open(wordlistFileName);
00169 
00170     std::mutex mlock;
00171     if(n<=50000000ull) return this->FastRandomWalkPartition(&mlock, &wordlist, n, minLen, maxLen,
      bFileIO, threads);
00172     else{
00173         int numberOfPartitions = n/50000000ull;
00174         for(int i=0;i<numberOfPartitions;i++)
00175             this->FastRandomWalkPartition(&mlock, &wordlist, 50000000ull, minLen, maxLen, bFileIO,
      threads);
00176     }
00177 
```

```
00178
00179 }
00180
00181
00182 void Markov::API::ModelMatrix::FastRandomWalkPartition(std::mutex *mlock, std::ofstream *wordlist,
      unsigned long int n, int minLen, int maxLen, bool bFileIO, int threads){
00183
00184     int iterationsPerThread = n/threads;
00185     int iterationsPerThreadCarryOver = n%threads;
00186
00187     std::vector<std::thread*> threadsV;
00188
00189     int id = 0;
00190     for(int i=0;i<threads;i++){
00191         threadsV.push_back(new std::thread(&Markov::API::ModelMatrix::FastRandomWalkThread, this,
      mlock, wordlist, iterationsPerThread, minLen, maxLen, id, bFileIO));
00192         id++;
00193     }
00194
00195     threadsV.push_back(new std::thread(&Markov::API::ModelMatrix::FastRandomWalkThread, this, mlock,
      wordlist, iterationsPerThreadCarryOver, minLen, maxLen, id, bFileIO));
00196
00197     for(int i=0;i<threads;i++){
00198         threadsV[i]->join();
00199     }
00200 }
```

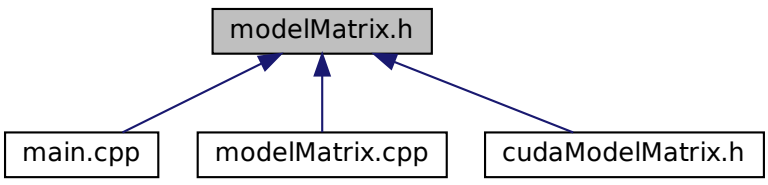
9.47 modelMatrix.h File Reference

```
#include "markovPasswords.h"
#include <mutex>
```

Include dependency graph for modelMatrix.h:



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



Classes

- class Markov::API::ModelMatrix

  Class to flatten and reduce Markov::Model to a Matrix.

Namespaces

- Markov

  Namespace for the markov-model related classes. Contains Model, Node and Edge classes.
- Markov::API

  Namespace for the MarkovPasswords API.

9.48 modelMatrix.h

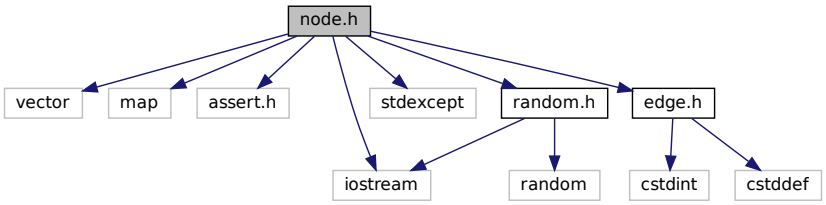
```
00001 #include "markovPasswords.h"
00002 #include <mutex>
00003
00004 namespace Markov::API{
00005
00006     /** @brief Class to flatten and reduce Markov::Model to a Matrix
00007      *
00008      * Matrix level operations can be used for Generation events, with a significant performance
      optimization at the cost of O(N) memory complexity (O(1) memory space for slow mode)
00009      *
00010      * To limit the maximum memory usage, each generation operation is partitioned into 50M chunks for
      allocation. Threads are sychronized and files are flushed every 50M operations.
00011      *
00012     */
00013     class ModelMatrix : public Markov::API::MarkovPasswords{
00014     public:
00015         ModelMatrix();
00016
00017         /** @brief Construct the related Matrix data for the model.
00018          *
00019          * This operation can be used after importing/training to allocate and populate the matrix
      content.
00020          *
00021          * this will initialize:
00022          * char** edgeMatrix -> a 2D array of mapping left and right connections of each edge.
00023          * long int **valueMatrix -> a 2D array representing the edge weights.
00024          * int matrixSize -> Size of the matrix, aka total number of nodes.
00025          * char* matrixIndex -> order of nodes in the model
00026          * long int *totalEdgeWeights -> total edge weights of each Node.
00027         */
00028         void ConstructMatrix();
00029
00030
00031         /** @brief Debug function to dump the model to a JSON file.
00032          *
00033          * Might not work 100%. Not meant for production use.
00034         */
00035         void DumpJSON();
00036
00037
00038         /** @brief Random walk on the Matrix-reduced Markov::Model
00039          *
00040          * This has an O(N) Memory complexity. To limit the maximum usage, requests with n>50M are
      partitioned using Markov::API::ModelMatrix::FastRandomWalkPartition.
00041          *
00042          * If n>50M, threads are going to be synced, files are going to be flushed, and buffers will
      be reallocated every 50M generations.
00043          * This comes at a minor performance penalty.
00044          *
00045          * While it has the same functionality, this operation reduces
      Markov::API::MarkovPasswords::Generate runtime by %96.5
00046          *
00047          * This function has deprecated Markov::API::MarkovPasswords::Generate, and will eventually
      replace it.
00048          *
00049          * @param n - Number of passwords to generate.
00050          * @param wordlistFileName - Filename to write to
00051          * @param minLen - Minimum password length to generate
00052          * @param maxLen - Maximum password length to generate
00053          * @param threads - number of OS threads to spawn
00054          * @param bFileIO - If false, filename will be ignored and will output to stdout.
00055          *
00056          *
00057          * @code{.cpp}
00058          * Markov::API::ModelMatrix mp;
```

```
00059          * mp.Import("models/finished.mdl");
00060          * mp.FastRandomWalk(50000000,"./wordlist.txt",6,12,25, true);
00061          * @endcode
00062          *
00063         */
00064         void FastRandomWalk(unsigned long int n, const char* wordlistFileName, int minLen=6, int
      maxLen=12, int threads=20, bool bFileIO=true);
00065
00066     protected:
00067
00068         /** @brief A single partition of FastRandomWalk event
00069          *
00070          * Since FastRandomWalk has to allocate its output buffer before operation starts and writes
      data in chunks,
00071          * large n parameters would lead to huge memory allocations.
00072          * @b Without @b Partitioning:
00073          * - 50M results 12 characters max -> 550 Mb Memory allocation
00074          *
00075          * - 5B results  12 characters max -> 55 Gb Memory allocation
00076          *
00077          * - 50B results 12 characters max -> 550GB Memory allocation
00078          *
00079          * Instead, FastRandomWalk is partitioned per 50M generations to limit the top memory need.
00080          *
00081          * @param mlock - mutex lock to distribute to child threads
00082          * @param wordlist - Reference to the wordlist file to write to
00083          * @param n - Number of passwords to generate.
00084          * @param wordlistFileName - Filename to write to
00085          * @param minLen - Minimum password length to generate
00086          * @param maxLen - Maximum password length to generate
00087          * @param threads - number of OS threads to spawn
00088          * @param bFileIO - If false, filename will be ignored and will output to stdout.
00089          *
00090          *
00091         */
00092         void FastRandomWalkPartition(std::mutex *mlock, std::ofstream *wordlist, unsigned long int n,
      int minLen, int maxLen, bool bFileIO, int threads);
00093
00094         /** @brief A single thread of a single partition of FastRandomWalk
00095          *
00096          * A FastRandomWalkPartition will initiate as many of this function as requested.
00097          *
00098          * This function contains the bulk of the generation algorithm.
00099          *
00100          * @param mlock - mutex lock to distribute to child threads
00101          * @param wordlist - Reference to the wordlist file to write to
00102          * @param n - Number of passwords to generate.
00103          * @param wordlistFileName - Filename to write to
00104          * @param minLen - Minimum password length to generate
00105          * @param maxLen - Maximum password length to generate
00106          * @param id - @b DEPRECATED Thread id - No longer used
00107          * @param bFileIO - If false, filename will be ignored and will output to stdout.
00108          *
00109          *
00110         */
00111         void FastRandomWalkThread(std::mutex *mlock, std::ofstream *wordlist, unsigned long int n, int
      minLen, int maxLen, int id, bool bFileIO);
00112
00113         /**
00114             @brief 2-D Character array for the edge Matrix (The characters of Nodes)
00115         */
00116         char** edgeMatrix;
00117
00118         /**
00119             @brief 2-d Integer array for the value Matrix (For the weights of  Edges)
00120         */
00121         long int **valueMatrix;
00122
00123         /**
00124             @brief to hold Matrix size
00125         */
00126         int matrixSize;
00127
00128         /**
00129             @brief to hold the Matrix index (To hold the orders of 2-D arrays')
00130         */
00131         char* matrixIndex;
00132
00133         /**
00134             @brief Array of the Total Edge Weights
00135         */
00136         long int *totalEdgeWeights;
00137     };
00138
00139
00140
00141 };
```

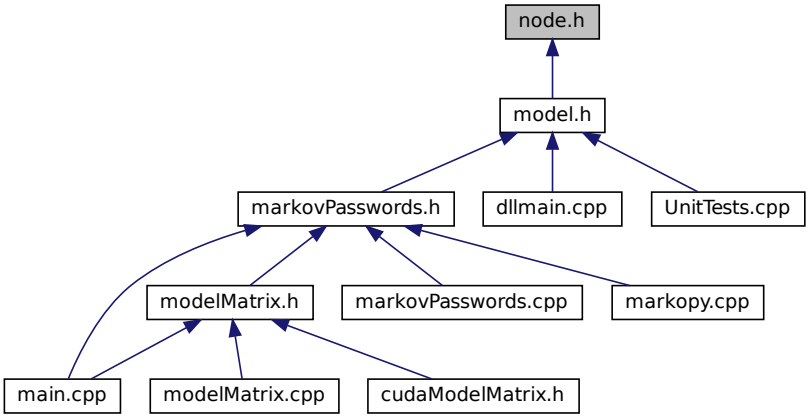
9.49 node.h File Reference

```
#include <vector>
#include <map>
#include <assert.h>
#include <iostream>
#include <stdexcept>
#include "edge.h"
#include "random.h"
```

Include dependency graph for node.h:



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



Classes

- class Markov::Node< storageType >

  A node class that for the vertices of model. Connected with eachother using Edge.

Namespaces

- Markov

  Namespace for the markov-model related classes. Contains Model, Node and Edge classes.

9.50 node.h

```
00001 #pragma once
00002 #include <vector>
```

```
00003 #include <map>
00004 #include <assert.h>
00005 #include <iostream>
00006 #include <stdexcept> // To use runtime_error
00007 #include "edge.h"
00008 #include "random.h"
00009 namespace Markov {
00010
00011     /** @brief A node class that for the vertices of model. Connected with eachother using Edge
00012     *
00013     * This class will later be templated to accept other data types than char*.
00014     */
00015     template <typename storageType>
00016     class Node {
00017     public:
00018
00019         /** @brief Default constructor. Creates an empty Node.
00020         */
00021         Node<storageType>();
00022
00023         /** @brief Constructor. Creates a Node with no edges and with given NodeValue.
00024          * @param _value - Nodes character representation.
00025          *
00026          * @b Example @b Use: Construct nodes
00027          * @code{.cpp}
00028          * Markov::Node<unsigned char>* LeftNode = new Markov::Node<unsigned char>('l');
00029          * Markov::Node<unsigned char>* RightNode = new Markov::Node<unsigned char>('r');
00030          * @endcode
00031         */
00032         Node<storageType>(storageType _value);
00033
00034         /** @brief Link this node with another, with this node as its source.
00035          *
00036          * Creates a new Edge.
00037          * @param target - Target node which will be the RightNode() of new edge.
00038          * @return A new node with LeftNode as this, and RightNode as parameter target.
00039          *
00040          * @b Example @b Use: Construct nodes
00041          * @code{.cpp}
00042          * Markov::Node<unsigned char>* LeftNode = new Markov::Node<unsigned char>('l');
00043          * Markov::Node<unsigned char>* RightNode = new Markov::Node<unsigned char>('r');
00044          * Markov::Edge<unsigned char>* e = LeftNode->Link(RightNode);
00045          * @endcode
00046         */
00047         Edge<storageType>* Link(Node<storageType>*);
00048
00049         /** @brief Link this node with another, with this node as its source.
00050          *
00051          * *DOES NOT* create a new Edge.
00052          * @param Edge - Edge that will accept this node as its LeftNode.
00053          * @return the same edge as parameter target.
00054          *
00055          * @b Example @b Use: Construct and link nodes
00056          * @code{.cpp}
00057          * Markov::Node<unsigned char>* LeftNode = new Markov::Node<unsigned char>('l');
00058          * Markov::Node<unsigned char>* RightNode = new Markov::Node<unsigned char>('r');
00059          * Markov::Edge<unsigned char>* e = LeftNode->Link(RightNode);
00060          * LeftNode->Link(e);
00061          * @endcode
00062         */
00063         Edge<storageType>* Link(Edge<storageType>*);
00064
00065         /** @brief Chose a random node from the list of edges, with regards to its EdgeWeight, and
      TraverseNode to that.
00066          *
00067          * This operation is done by generating a random number in range of 0-this.total_edge_weights,
      and then iterating over the list of edges.
00068          * At each step, EdgeWeight of the edge is subtracted from the random number, and once it is
      0, next node is selected.
00069          * @return Node that was chosen at EdgeWeight biased random.
00070          *
00071          * @b Example @b Use: Use randomNext to do a random walk on the model
00072          * @code{.cpp}
00073          *  char* buffer[64];
00074          *  Markov::Model<char> model;
00075          *  model.Import("model.mdl");
00076          *  Markov::Node<char>* n = model.starterNode;
00077          *  int len = 0;
00078          *  Markov::Node<char>* temp_node;
00079          *  while (true) {
00080          *      temp_node = n->RandomNext(randomEngine);
00081          *      if (len >= maxSetting) {
00082          *          break;
00083          *      }
00084          *      else if ((temp_node == NULL) && (len < minSetting)) {
00085          *          continue;
00086          *      }
```

```
00087          *
00088          *      else if (temp_node == NULL){
00089          *          break;
00090          *      }
00091          *
00092          *      n = temp_node;
00093          *
00094          *      buffer[len++] = n->NodeValue();
00095          *  }
00096          * @endcode
00097          */
00098         Node<storageType>* RandomNext(Markov::Random::RandomEngine* randomEngine);
00099
00100         /** @brief Insert a new edge to the this.edges.
00101          * @param edge - New edge that will be inserted.
00102          * @return true if insertion was successful, false if it fails.
00103          *
00104          * @b Example @b Use: Construct and update edges
00105          *
00106          * @code{.cpp}
00107          * Markov::Node<unsigned char>* src = new Markov::Node<unsigned char>('a');
00108          * Markov::Node<unsigned char>* target1 = new Markov::Node<unsigned char>('b');
00109          * Markov::Node<unsigned char>* target2 = new Markov::Node<unsigned char>('c');
00110          * Markov::Edge<unsigned char>* e1 = new Markov::Edge<unsigned char>(src, target1);
00111          * Markov::Edge<unsigned char>* e2 = new Markov::Edge<unsigned char>(src, target2);
00112          * e1->AdjustEdge(25);
00113          * src->UpdateEdges(e1);
00114          * e2->AdjustEdge(30);
00115          * src->UpdateEdges(e2);
00116          * @endcode
00117          */
00118         bool UpdateEdges(Edge<storageType>*);
00119
00120         /** @brief Find an edge with its character representation.
00121          * @param repr - character NodeValue of the target node.
00122          * @return Edge that is connected between this node, and the target node.
00123          *
00124          * @b Example @b Use: Construct and update edges
00125          *
00126          * @code{.cpp}
00127          * Markov::Node<unsigned char>* src = new Markov::Node<unsigned char>('a');
00128          * Markov::Node<unsigned char>* target1 = new Markov::Node<unsigned char>('b');
00129          * Markov::Node<unsigned char>* target2 = new Markov::Node<unsigned char>('c');
00130          * Markov::Edge<unsigned char>* res = NULL;
00131          * src->Link(target1);
00132          * src->Link(target2);
00133          * res = src->FindEdge('b');
00134          *
00135          * @endcode
00136          *
00137          */
00138         Edge<storageType>* FindEdge(storageType repr);
00139
00140         /** @brief Find an edge with its pointer. Avoid unless neccessary because comptutational cost
      of find by character is cheaper (because of std::map)
00141         * @param target - target node.
00142         * @return Edge that is connected between this node, and the target node.
00143         */
00144         Edge<storageType>* FindEdge(Node<storageType>* target);
00145
00146         /** @brief Return character representation of this node.
00147         * @return character representation at _value.
00148         */
00149         inline unsigned char NodeValue();
00150
00151         /** @brief Change total weights with offset
00152          * @param offset to adjust the vertice weight with
00153         */
00154         void UpdateTotalVerticeWeight(long int offset);
00155
00156         /** @brief return edges
00157         */
00158         inline std::map<storageType, Edge<storageType>*>* Edges();
00159
00160         /** @brief return total edge weights
00161         */
00162         inline long int TotalEdgeWeights();
00163
00164
00165         std::vector<Edge<storageType>*> edgesV;
00166     private:
00167
00168         /**
00169             @brief Character representation of this node. 0 for starter, 0xff for terminator.
00170         */
00171         storageType _value;
00172
```

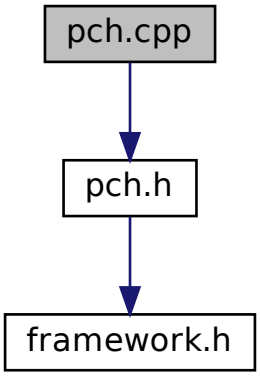
```
00173         /**
00174             @brief Total weights of the vertices, required by RandomNext
00175         */
00176         long int total_edge_weights;
00177 
00178         /**
00179             @brief A map of all edges connected to this node, where this node is at the LeftNode.
00180             * Map is indexed by unsigned char, which is the character representation of the node.
00181         */
00182         std::map<storageType, Edge<storageType>*> edges;
00183     };
00184 };
00185 
00186 
00187 
00188 
00189 
00190 
00191 
00192 
00193 
00194 template <typename storageType>
00195 Markov::Node<storageType>::Node(storageType _value) {
00196     this->_value = _value;
00197     this->total_edge_weights = 0L;
00198 };
00199 
00200 template <typename storageType>
00201 Markov::Node<storageType>::Node() {
00202     this->_value = 0;
00203     this->total_edge_weights = 0L;
00204 };
00205 
00206 template <typename storageType>
00207 inline unsigned char Markov::Node<storageType>::NodeValue() {
00208     return _value;
00209 }
00210 
00211 template <typename storageType>
00212 Markov::Edge<storageType>* Markov::Node<storageType>::Link(Markov::Node<storageType>* n) {
00213     Markov::Edge<storageType>* v = new Markov::Edge<storageType>(this, n);
00214     this->UpdateEdges(v);
00215     return v;
00216 }
00217 
00218 template <typename storageType>
00219 Markov::Edge<storageType>* Markov::Node<storageType>::Link(Markov::Edge<storageType>* v) {
00220     v->SetLeftEdge(this);
00221     this->UpdateEdges(v);
00222     return v;
00223 }
00224 
00225 template <typename storageType>
00226 Markov::Node<storageType>* Markov::Node<storageType>::RandomNext(Markov::Random::RandomEngine*
       randomEngine) {
00227 
00228     //get a random NodeValue in range of total_vertice_weight
00229     long int selection = randomEngine->random() %
       this->total_edge_weights;//distribution()(generator());// distribution(generator);
00230     //make absolute, no negative modulus values wanted
00231     //selection = (selection >= 0) ? selection : (selection + this->total_edge_weights);
00232     for(int i=0;i<this->edgesV.size();i++){
00233         selection -= this->edgesV[i]->EdgeWeight();
00234         if (selection < 0) return this->edgesV[i]->TraverseNode();
00235     }
00236 
00237     //if this assertion is reached, it means there is an implementation error above
00238     std::cout << "This should never be reached (node failed to walk to next)\n"; //cant assert from
       child thread
00239     assert(true && "This should never be reached (node failed to walk to next)");
00240     return NULL;
00241 }
00242 
00243 template <typename storageType>
00244 bool Markov::Node<storageType>::UpdateEdges(Markov::Edge<storageType>* v) {
00245     this->edges.insert({ v->RightNode()->NodeValue(), v });
00246     this->edgesV.push_back(v);
00247     //this->total_edge_weights += v->EdgeWeight();
00248     return v->TraverseNode();
00249 }
00250 
00251 template <typename storageType>
00252 Markov::Edge<storageType>* Markov::Node<storageType>::FindEdge(storageType repr) {
00253     auto e = this->edges.find(repr);
00254     if (e == this->edges.end()) return NULL;
00255     return e->second;
00256 };
```

```
00257
00258 template <typename storageType>
00259 void Markov::Node<storageType>::UpdateTotalVerticeWeight(long int offset) {
00260     this->total_edge_weights += offset;
00261 }
00262
00263 template <typename storageType>
00264 inline std::map<storageType, Markov::Edge<storageType>*>* Markov::Node<storageType>::Edges() {
00265     return &(this->edges);
00266 }
00267
00268 template <typename storageType>
00269 inline long int Markov::Node<storageType>::TotalEdgeWeights() {
00270     return this->total_edge_weights;
00271 }
```

9.51 pch.cpp File Reference

`#include "pch.h"`

Include dependency graph for MarkovModel/src/pch.cpp:



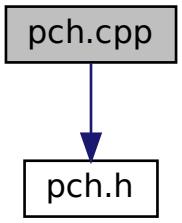
9.52 MarkovModel/src/pch.cpp

```
00001 // pch.cpp: source file corresponding to the pre-compiled header
00002
00003 #include "pch.h"
00004
00005 // When you are using pre-compiled headers, this source file is necessary for compilation to succeed.
```

9.53 pch.cpp File Reference

`#include "pch.h"`

Include dependency graph for UnitTests/pch.cpp:

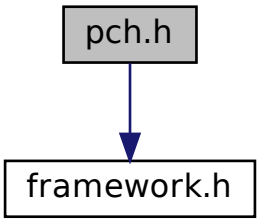


9.54 UnitTests/pch.cpp

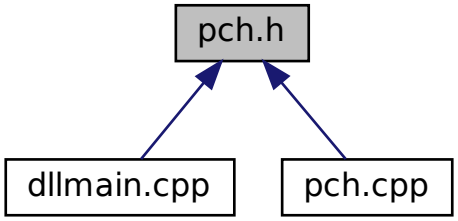
```
00001 // pch.cpp: source file corresponding to the pre-compiled header
00002
00003 #include "pch.h"
00004
00005 // When you are using pre-compiled headers, this source file is necessary for compilation to succeed.
```

9.55 pch.h File Reference

`#include "framework.h"`
Include dependency graph for MarkovModel/src/pch.h:



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

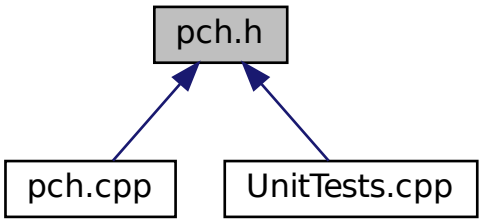


9.56 MarkovModel/src/pch.h

```
00001 // pch.h: This is a precompiled header file.
00002 // Files listed below are compiled only once, improving build performance for future builds.
00003 // This also affects IntelliSense performance, including code completion and many code browsing
       features.
00004 // However, files listed here are ALL re-compiled if any one of them is updated between builds.
00005 // Do not add files here that you will be updating frequently as this negates the performance
       advantage.
00006
00007 #ifndef PCH_H
00008 #define PCH_H
00009
00010 // add headers that you want to pre-compile here
00011 #include "framework.h"
00012
00013 #endif //PCH_H
```

9.57 pch.h File Reference

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



9.58 UnitTests/pch.h

```
00001 // pch.h: This is a precompiled header file.
00002 // Files listed below are compiled only once, improving build performance for future builds.
00003 // This also affects IntelliSense performance, including code completion and many code browsing
       features.
00004 // However, files listed here are ALL re-compiled if any one of them is updated between builds.
00005 // Do not add files here that you will be updating frequently as this negates the performance
       advantage.
```

```
00006
00007 #ifndef PCH_H
00008 #define PCH_H
00009
00010 // add headers that you want to pre-compile here
00011
00012 #endif //PCH_H
```

9.59 random-model.py File Reference

Namespaces

- random-model
- random

Variables

- random-model.alphabet = string.printable

  password alphabet

- random-model.f = open('../../models/random.mdl', "wb")

  output file handle

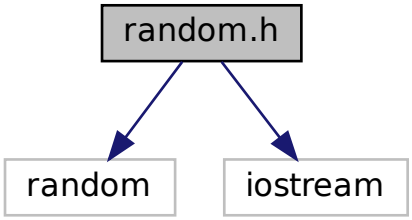
9.60 random-model.py

```
00001 #!/usr/bin/python3
00002 """
00003   python script for generating a 2gram model
00004 """
00005
00006 import string
00007 import re
00008
00009
00010 alphabet = string.printable
00011 alphabet = re.sub('\s', '', alphabet)
00012 print(f"alphabet={alphabet}")
00013 #exit()
00014
00015
00016 f = open('../../models/random.mdl', "wb")
00017 #tie start nodes
00018 for sym in alphabet:
00019     f.write(b"\x00,1," + bytes(sym, encoding='ascii') + b"\n")
00020
00021 #tie terminator nodes
00022 for sym in alphabet:
00023     f.write(bytes(sym, encoding='ascii')+ b",1,\xff\n")
00024
00025 #tie internals
00026 for src in alphabet:
00027     for target in alphabet:
00028         f.write(bytes(src, encoding='ascii') + b",1," + bytes(target, encoding='ascii') + b"\n")
```

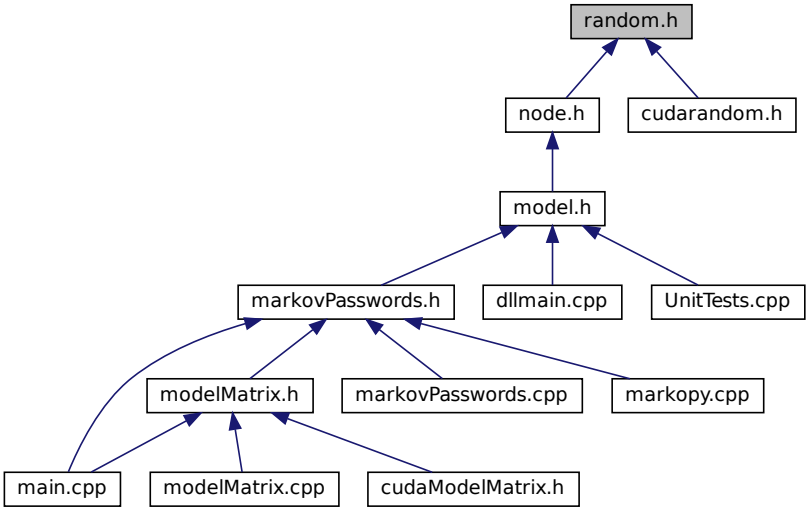
9.61 random.h File Reference

```
#include <random>
#include <iostream>
```

Include dependency graph for random.h:



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



Classes

- class Markov::Random::RandomEngine

  An abstract class for Random Engine.
- class Markov::Random::DefaultRandomEngine

  Implementation using Random.h default random engine.
- class Markov::Random::Marsaglia

  Implementation of Marsaglia Random Engine.
- class Markov::Random::Mersenne

  Implementation of Mersenne Twister Engine.

Namespaces

- Markov

  Namespace for the markov-model related classes. Contains Model, Node and Edge classes.
- Markov::Random

Objects related to RNG.

9.62 random.h

```
00001
00002 #pragma once
00003 #include <random>
00004 #include <iostream>
00005
00006 /**
00007     @brief Objects related to RNG
00008 */
00009 namespace Markov::Random{
00010
00011     /** @brief An abstract class for Random Engine
00012      *
00013      * This class is used for generating random numbers, which are used for random walking on the
      graph.
00014      *
00015      * Main reason behind allowing different random engines is that some use cases may favor
      performance,
00016      * while some favor good random.
00017      *
00018      * Mersenne can be used for truer random, while Marsaglia can be used for deterministic but fast
      random.
00019      *
00020     */
00021     class RandomEngine{
00022     public:
00023         virtual inline unsigned long random() = 0;
00024     };
00025
00026
00027
00028     /** @brief Implementation using Random.h default random engine
00029      *
00030      * This engine is also used by other engines for seeding.
00031      *
00032      *
00033      * @b Example @b Use: Using Default Engine with RandomWalk
00034      * @code{.cpp}
00035      * Markov::Model<char> model;
00036      * Model.import("model.mdl");
00037      * char* res = new char[11];
00038      * Markov::Random::DefaultRandomEngine randomEngine;
00039      * for (int i = 0; i < 10; i++) {
00040      *      this->RandomWalk(&randomEngine, 5, 10, res);
00041      *      std::cout « res « "\n";
00042      *  }
00043      * @endcode
00044      *
00045      * @b Example @b Use: Generating a random number with Marsaglia Engine
00046      * @code{.cpp}
00047      * Markov::Random::DefaultRandomEngine de;
00048      * std::cout « de.random();
00049      * @endcode
00050      *
00051     */
00052     class DefaultRandomEngine : public RandomEngine{
00053     public:
00054         /** @brief Generate Random Number
00055         * @return random number in long range.
00056         */
00057         inline unsigned long random(){
00058             return this->distribution()(this->generator());
00059         }
00060     protected:
00061
00062         /** @brief Default random device for seeding
00063         *
00064         */
00065         inline std::random_device& rd() {
00066             static std::random_device _rd;
00067             return _rd;
00068         }
00069
00070         /** @brief Default random engine for seeding
00071         *
00072         */
00073         inline std::default_random_engine& generator() {
00074             static std::default_random_engine _generator(rd()());
00075             return _generator;
00076         }
00077
00078         /** @brief Distribution schema for seeding.
```

```
00079         *
00080         */
00081         inline std::uniform_int_distribution<long long unsigned>& distribution() {
00082             static std::uniform_int_distribution<long long unsigned> _distribution(0, 0xffffFFFF);
00083             return _distribution;
00084         }
00085
00086     };
00087
00088
00089     /** @brief Implementation of Marsaglia Random Engine
00090      *
00091      * This is an implementation of Marsaglia Random engine, which for most use cases is a better fit
    than other solutions.
00092      * Very simple mathematical formula to generate pseudorandom integer, so its crazy fast.
00093      *
00094      * This implementation of the Marsaglia Engine is seeded by random.h default random engine.
00095      * RandomEngine is only seeded once so its not a performance issue.
00096      *
00097      * @b Example @b Use: Using Marsaglia Engine with RandomWalk
00098      * @code{.cpp}
00099      * Markov::Model<char> model;
00100     * Model.import("model.mdl");
00101     * char* res = new char[11];
00102     * Markov::Random::Marsaglia MarsagliaRandomEngine;
00103     * for (int i = 0; i < 10; i++) {
00104     *      this->RandomWalk(&MarsagliaRandomEngine, 5, 10, res);
00105     *      std::cout « res « "\n";
00106     *  }
00107     * @endcode
00108     *
00109     * @b Example @b Use: Generating a random number with Marsaglia Engine
00110     * @code{.cpp}
00111     * Markov::Random::Marsaglia me;
00112     * std::cout « me.random();
00113     * @endcode
00114     *
00115     */
00116     class Marsaglia : public DefaultRandomEngine{
00117     public:
00118
00119         /** @brief Construct Marsaglia Engine
00120         *
00121         * Initialize x,y and z using the default random engine.
00122         */
00123         Marsaglia(){
00124             this->x = this->distribution()(this->generator());
00125             this->y = this->distribution()(this->generator());
00126             this->z = this->distribution()(this->generator());
00127             //std::cout « "x: " « x « ", y: " « y « ", z: " « z « "\n";
00128         }
00129
00130
00131     inline unsigned long random(){
00132         unsigned long t;
00133         x ^= x « 16;
00134         x ^= x » 5;
00135         x ^= x « 1;
00136
00137         t = x;
00138         x = y;
00139         y = z;
00140         z = t ^ x ^ y;
00141
00142         return z;
00143     }
00144
00145
00146         unsigned long x;
00147         unsigned long y;
00148         unsigned long z;
00149     };
00150
00151
00152     /** @brief Implementation of Mersenne Twister Engine
00153      *
00154      * This is an implementation of Mersenne Twister Engine, which is slow but is a good
    implementation for high entropy pseudorandom.
00155      *
00156      *
00157      * @b Example @b Use: Using Mersenne Engine with RandomWalk
00158      * @code{.cpp}
00159      * Markov::Model<char> model;
00160      * Model.import("model.mdl");
00161      * char* res = new char[11];
00162      * Markov::Random::Mersenne MersenneTwisterEngine;
00163      * for (int i = 0; i < 10; i++) {
```

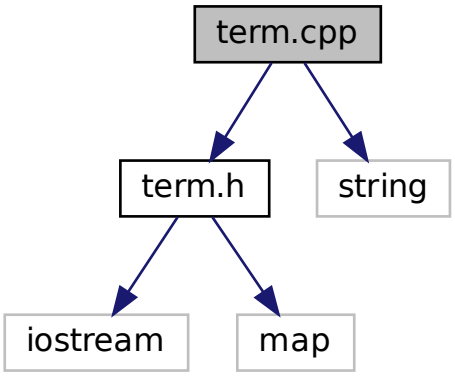
```
00164      *      this->RandomWalk(&MersenneTwisterEngine, 5, 10, res);
00165      *      std::cout « res « "\n";
00166      *  }
00167      * @endcode
00168      *
00169      * @b Example @b Use: Generating a random number with Marsaglia Engine
00170      * @code{.cpp}
00171      * Markov::Random::Mersenne me;
00172      * std::cout « me.random();
00173      * @endcode
00174      *
00175     */
00176     class Mersenne : public DefaultRandomEngine{
00177
00178     };
00179
00180
00181 };
```

9.63 README.md File Reference

9.64 term.cpp File Reference

```
#include "term.h"
#include <string>
```

Include dependency graph for term.cpp:



Functions

- std::ostream & operator<< (std::ostream &os, const Terminal::color &c)

9.64.1 Function Documentation

9.64.1.1 operator<<()

```
std::ostream& operator<< (
            std::ostream & os,
            const Markov::API::CLI::Terminal::color & c )
```

overload for std::cout.

Definition at line 60 of file term.cpp.

```
00060                                                    {
```

```
00061     char buf[6];
00062     sprintf(buf,"%d",Terminal::colormap.find(c)->second);
00063     os « "\e[1;" « buf « "m";
00064     return os;
00065 }
```

References Markov::API::CLI::Terminal::colormap.

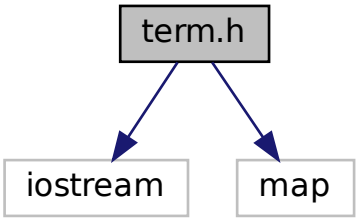
9.65 term.cpp

```
00001 #pragma once
00002 #include "term.h"
00003 #include <string>
00004
00005 using namespace Markov::API::CLI;
00006
00007 //Windows text processing is different from unix systems, so use windows header and text attributes
00008 #ifdef _WIN32
00009
00010 HANDLE Terminal::_stdout;
00011 HANDLE Terminal::_stderr;
00012
00013 std::map<Terminal::color, DWORD> Terminal::colormap = {
00014     {Terminal::color::BLACK, 0},
00015     {Terminal::color::BLUE, 1},
00016     {Terminal::color::GREEN, 2},
00017     {Terminal::color::CYAN, 3},
00018     {Terminal::color::RED, 4},
00019     {Terminal::color::MAGENTA, 5},
00020     {Terminal::color::BROWN, 6},
00021     {Terminal::color::LIGHTGRAY, 7},
00022     {Terminal::color::DARKGRAY, 8},
00023     {Terminal::color::YELLOW, 14},
00024     {Terminal::color::WHITE, 15},
00025     {Terminal::color::RESET, 15},
00026 };
00027
00028
00029 Terminal::Terminal() {
00030     Terminal::_stdout = GetStdHandle(STD_OUTPUT_HANDLE);
00031     Terminal::_stderr = GetStdHandle(STD_ERROR_HANDLE);
00032 }
00033
00034 std::ostream& operator«(std::ostream& os, const Terminal::color& c) {
00035     SetConsoleTextAttribute(Terminal::_stdout, Terminal::colormap.find(c)->second);
00036     return os;
00037 }
00038
00039 #else
00040
00041 std::map<Terminal::color, int> Terminal::colormap = {
00042     {Terminal::color::BLACK, 30},
00043     {Terminal::color::BLUE, 34},
00044     {Terminal::color::GREEN, 32},
00045     {Terminal::color::CYAN, 36},
00046     {Terminal::color::RED, 31},
00047     {Terminal::color::MAGENTA, 35},
00048     {Terminal::color::BROWN, 0},
00049     {Terminal::color::LIGHTGRAY, 0},
00050     {Terminal::color::DARKGRAY, 0},
00051     {Terminal::color::YELLOW, 33},
00052     {Terminal::color::WHITE, 37},
00053     {Terminal::color::RESET, 0},
00054 };
00055
00056 Terminal::Terminal() {
00057     /*this->;*/
00058 }
00059
00060 std::ostream& operator«(std::ostream& os, const Terminal::color& c) {
00061     char buf[6];
00062     sprintf(buf,"%d",Terminal::colormap.find(c)->second);
00063     os « "\e[1;" « buf « "m";
00064     return os;
00065 }
00066
00067
00068
00069
00070 #endif
```

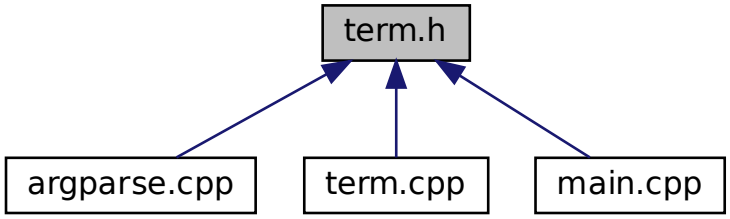
9.66 term.h File Reference

```
#include <iostream>
#include <map>
```

Include dependency graph for term.h:



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



Classes

- class Markov::API::CLI::Terminal

  pretty colors for Terminal. Windows Only.

Namespaces

- Markov

  Namespace for the markov-model related classes. Contains Model, Node and Edge classes.
- Markov::API

  Namespace for the MarkovPasswords API.
- Markov::API::CLI

  Structure to hold parsed cli arguements.

Macros

- #define TERM_FAIL "[" << Markov::API::CLI::Terminal::color::RED << "+" << Markov::API::CLI::Terminal::color::RESET << "] "

- #define TERM_INFO "[" << Markov::API::CLI::Terminal::color::BLUE << "+" << Markov::API::CLI::Terminal::color::RESET << "] "
- #define TERM_WARN "[" << Markov::API::CLI::Terminal::color::YELLOW << "+" << Markov::API::CLI::Terminal::color::RESE << "] "
- #define TERM_SUCC "[" << Markov::API::CLI::Terminal::color::GREEN << "+" << Markov::API::CLI::Terminal::color::RESET << "] "

Functions

- std::ostream & Markov::API::CLI::operator<< (std::ostream &os, const Markov::API::CLI::Terminal::color &c)

9.66.1 Macro Definition Documentation

9.66.1.1 TERM_FAIL

```
#define TERM_FAIL "[" << Markov::API::CLI::Terminal::color::RED << "+" << Markov::API::CLI::Terminal::color
<< "] "
```

Definition at line 10 of file term.h.

9.66.1.2 TERM_INFO

```
#define TERM_INFO "[" << Markov::API::CLI::Terminal::color::BLUE << "+" << Markov::API::CLI::Terminal::colo
<< "] "
```

Definition at line 11 of file term.h.

9.66.1.3 TERM_SUCC

```
#define TERM_SUCC "[" << Markov::API::CLI::Terminal::color::GREEN << "+" << Markov::API::CLI::Terminal::col
<< "] "
```

Definition at line 13 of file term.h.

9.66.1.4 TERM_WARN

```
#define TERM_WARN "[" << Markov::API::CLI::Terminal::color::YELLOW << "+" << Markov::API::CLI::Terminal::co
<< "] "
```

Definition at line 12 of file term.h.

9.67 term.h

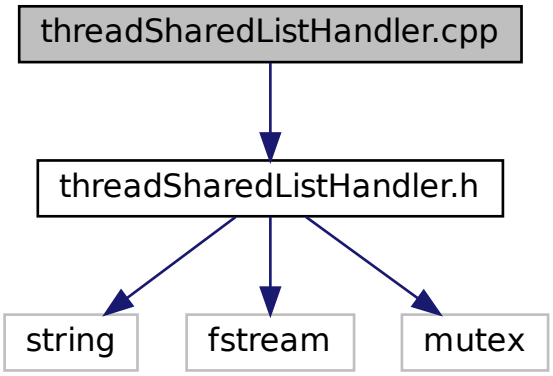
```
00001 #pragma once
00002
00003 #ifdef _WIN32
00004 #include <Windows.h>
00005 #endif
00006
00007 #include <iostream>
00008 #include <map>
00009
00010 #define TERM_FAIL "[" « Markov::API::CLI::Terminal::color::RED « "+" «
       Markov::API::CLI::Terminal::color::RESET « "] "
00011 #define TERM_INFO "[" « Markov::API::CLI::Terminal::color::BLUE « "+" «
       Markov::API::CLI::Terminal::color::RESET « "] "
00012 #define TERM_WARN "[" « Markov::API::CLI::Terminal::color::YELLOW « "+" «
       Markov::API::CLI::Terminal::color::RESET « "] "
00013 #define TERM_SUCC "[" « Markov::API::CLI::Terminal::color::GREEN « "+" «
       Markov::API::CLI::Terminal::color::RESET « "] "
00014
00015 namespace Markov::API::CLI{
00016     /** @brief pretty colors for Terminal. Windows Only.
00017     */
```

```
00018     class Terminal {
00019     public:
00020
00021         /** Default constructor.
00022         * Get references to stdout and stderr handles.
00023         */
00024         Terminal();
00025
00026         enum color { RESET, BLACK, RED, GREEN, YELLOW, BLUE, MAGENTA, CYAN, WHITE, LIGHTGRAY,
      DARKGRAY, BROWN };
00027         #ifdef _WIN32
00028         static HANDLE _stdout;
00029         static HANDLE _stderr;
00030         static std::map<Markov::API::CLI::Terminal::color, DWORD> colormap;
00031         #else
00032         static std::map<Markov::API::CLI::Terminal::color, int> colormap;
00033         #endif
00034
00035
00036
00037         static std::ostream endl;
00038
00039
00040     };
00041
00042     /** overload for std::cout.
00043     */
00044     std::ostream& operator«(std::ostream& os, const Markov::API::CLI::Terminal::color& c);
00045
00046 }
```

9.68 threadSharedListHandler.cpp File Reference

`#include "threadSharedListHandler.h"`

Include dependency graph for threadSharedListHandler.cpp:



9.69 threadSharedListHandler.cpp

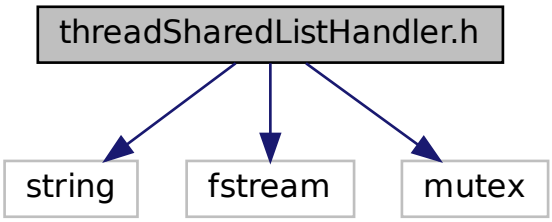
```
00001 #include "threadSharedListHandler.h"
00002
00003
00004 Markov::API::Concurrency::ThreadSharedListHandler::ThreadSharedListHandler(const char* filename){
00005     this->listfile;
00006     this->listfile.open(filename, std::ios_base::binary);
00007 }
00008
00009
00010 bool Markov::API::Concurrency::ThreadSharedListHandler::next(std::string* line){
00011     bool res = false;
00012     this->mlock.lock();
00013     res = (std::getline(this->listfile,*line,'\n'))? true : false;
```

```
00014     this->mlock.unlock();
00015
00016     return res;
00017 }
```

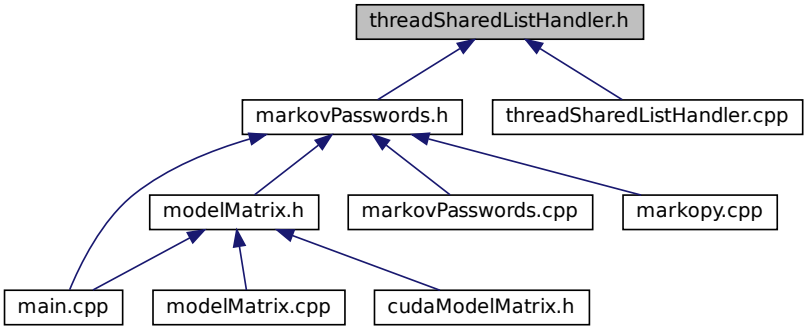
9.70 threadSharedListHandler.h File Reference

```
#include <string>
#include <fstream>
#include <mutex>
```

Include dependency graph for threadSharedListHandler.h:



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



Classes

- class Markov::API::Concurrency::ThreadSharedListHandler

  Simple class for managing shared access to file.

Namespaces

- Markov

  Namespace for the markov-model related classes. Contains Model, Node and Edge classes.
- Markov::API

  Namespace for the MarkovPasswords API.
- Markov::API::Concurrency

  Namespace for Concurrency related classes.

9.71 threadSharedListHandler.h

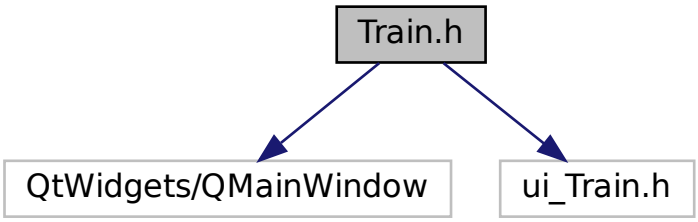
```
00001 #include <string>
00002 #include <fstream>
00003 #include <mutex>
00004
00005 /** @brief Namespace for Concurrency related classes
00006 */
00007 namespace Markov::API::Concurrency{
00008
00009 /** @brief Simple class for managing shared access to file
00010  *
00011  * This class maintains the handover of each line from a file to multiple threads.
00012  *
00013  * When two different threads try to read from the same file while reading a line isn't completed, it
      can have unexpected results.
00014  * Line might be split, or might be read twice.
00015  * This class locks the read action on the list until a line is completed, and then proceeds with the
      handover.
00016  *
00017 */
00018 class ThreadSharedListHandler{
00019 public:
00020     /** @brief Construct the Thread Handler with a filename
00021      *
00022      * Simply open the file, and initialize the locks.
00023      *
00024      * @b Example @b Use: Simple file read
00025      * @code{.cpp}
00026      * ThreadSharedListHandler listhandler("test.txt");
00027      * std::string line;
00028      * std::cout « listhandler->next(&line) « "\n";
00029      * @endcode
00030      *
00031      * @b Example @b Use: Example use case from MarkovPasswords showing multithreaded access
00032      * @code{.cpp}
00033      *  void MarkovPasswords::Train(const char* datasetFileName, char delimiter, int threads)   {
00034      *       ThreadSharedListHandler listhandler(datasetFileName);
00035      *       auto start = std::chrono::high_resolution_clock::now();
00036      *
00037      *       std::vector<std::thread*> threadsV;
00038      *       for(int i=0;i<threads;i++){
00039      *           threadsV.push_back(new std::thread(&MarkovPasswords::TrainThread, this, &listhandler,
      datasetFileName, delimiter));
00040      *       }
00041      *
00042      *      for(int i=0;i<threads;i++){
00043      *           threadsV[i]->join();
00044      *           delete threadsV[i];
00045      *       }
00046      *       auto finish = std::chrono::high_resolution_clock::now();
00047      *       std::chrono::duration<double> elapsed = finish - start;
00048      *       std::cout « "Elapsed time: " « elapsed.count() « " s\n";
00049      *
00050      *   }
00051      *
00052      *   void MarkovPasswords::TrainThread(ThreadSharedListHandler *listhandler, const char*
      datasetFileName, char delimiter){
00053      *       char format_str[] ="%ld,%s";
00054      *       format_str[2]=delimiter;
00055      *       std::string line;
00056      *       while (listhandler->next(&line)) {
00057      *           long int oc;
00058      *           if (line.size() > 100) {
00059      *               line = line.substr(0, 100);
00060      *           }
00061      *           char* linebuf = new char[line.length()+5];
00062      *           sscanf_s(line.c_str(), format_str, &oc, linebuf, line.length()+5);
00063      *           this->AdjustEdge((const char*)linebuf, oc);
00064      *           delete linebuf;
00065      *       }
00066      *   }
00067      * @endcode
00068      *
00069      * @param filename Filename for the file to manage.
00070     */
00071     ThreadSharedListHandler(const char* filename);
00072
00073     /** @brief Read the next line from the file.
00074      *
00075      * This action will be blocked until another thread (if any) completes the read operation on the
      file.
00076      *
00077      * @b Example @b Use: Simple file read
00078      * @code{.cpp}
00079      * ThreadSharedListHandler listhandler("test.txt");
00080      * std::string line;
```

```
00081      * std::cout « listhandler->next(&line) « "\n";
00082      * @endcode
00083      *
00084     */
00085     bool next(std::string* line);
00086
00087 private:
00088     std::ifstream listfile;
00089     std::mutex mlock;
00090 };
00091
00092 };
```

9.72 Train.h File Reference

```
#include <QtWidgets/QMainWindow>
#include "ui_Train.h"
```

Include dependency graph for Train.h:



Classes

- class Markov::GUI::Train

  QT Training page class.

Namespaces

- Markov

  Namespace for the markov-model related classes. Contains Model, Node and Edge classes.
- Markov::GUI

  namespace for MarkovPasswords API GUI wrapper

9.73 Train.h

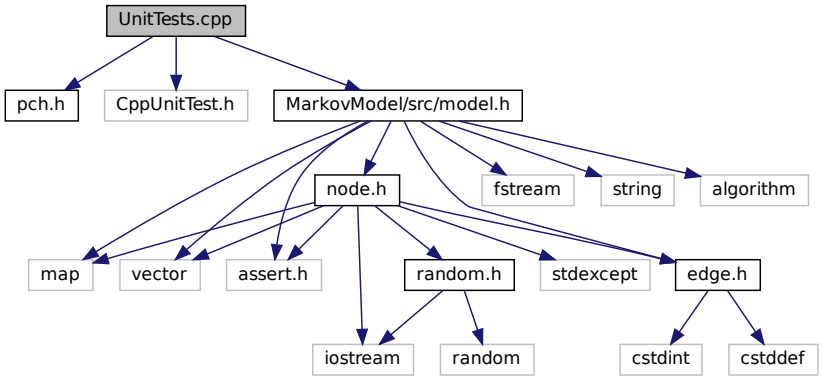
```
00001 #pragma once
00002 #include <QtWidgets/QMainWindow>
00003 #include "ui_Train.h"
00004
00005 namespace Markov::GUI{
00006
00007     /** @brief QT Training page class
00008     */
00009     class Train :public QMainWindow {
00010     Q_OBJECT
00011     public:
00012         Train(QWidget* parent = Q_NULLPTR);
00013
00014     private:
00015         Ui::Train ui;
00016
00017     public slots:
```

```
00018         void home();
00019         void train();
00020     };
00021 };
```

9.74 UnitTests.cpp File Reference

```
#include "pch.h"
#include "CppUnitTest.h"
#include "MarkovModel/src/model.h"
```

Include dependency graph for UnitTests.cpp:



Namespaces

- Testing

  Namespace for Microsoft Native Unit Testing Classes.
- Testing::MVP

  Testing Namespace for Minimal Viable Product.
- Testing::MVP::MarkovModel

  Testing Namespace for MVP MarkovModel.
- Testing::MVP::MarkovPasswords

  Testing namespace for MVP MarkovPasswords.
- Testing::MarkovModel

  Testing namespace for MarkovModel.
- Testing::MarkovPasswords

  Testing namespace for MarkovPasswords.

Functions

- Testing::MVP::MarkovModel::TEST_CLASS (Edge)

  Test class for minimal viable Edge.
- Testing::MVP::MarkovModel::TEST_CLASS (Node)

  Test class for minimal viable Node.
- Testing::MVP::MarkovModel::TEST_CLASS (Model)

  Test class for minimal viable Model.
- Testing::MVP::MarkovPasswords::TEST_CLASS (ArgParser)

  Test Class for Argparse class.
- Testing::MarkovModel::TEST_CLASS (Edge)

Test class for rest of Edge cases.

- Testing::MarkovModel::TEST_CLASS (Node)

  Test class for rest of Node cases.

- Testing::MarkovModel::TEST_CLASS (Model)

  Test class for rest of model cases.

9.75 UnitTests.cpp

```
00001 #include "pch.h"
00002 #include "CppUnitTest.h"
00003 #include "MarkovModel/src/model.h"
00004
00005 using namespace Microsoft::VisualStudio::CppUnitTestFramework;
00006
00007
00008 /** @brief Namespace for Microsoft Native Unit Testing Classes
00009 */
00010 namespace Testing {
00011
00012     /** @brief Testing Namespace for Minimal Viable Product
00013     */
00014     namespace MVP {
00015         /** @brief Testing Namespace for MVP MarkovModel
00016         */
00017         namespace MarkovModel
00018         {
00019             /** @brief Test class for minimal viable Edge
00020             */
00021             TEST_CLASS(Edge)
00022             {
00023             public:
00024
00025                 /** @brief test default constructor
00026                 */
00027                 TEST_METHOD(default_constructor) {
00028                     Markov::Edge<unsigned char>* e = new Markov::Edge<unsigned char>;
00029                     Assert::IsNull(e->LeftNode());
00030                     Assert::IsNull(e->RightNode());
00031                     delete e;
00032                 }
00033
00034                 /** @brief test linked constructor with two nodes
00035                 */
00036                 TEST_METHOD(linked_constructor) {
00037                     Markov::Node<unsigned char>* LeftNode = new Markov::Node<unsigned char>('l');
00038                     Markov::Node<unsigned char>* RightNode = new Markov::Node<unsigned char>('r');
00039                     Markov::Edge<unsigned char>* e = new Markov::Edge<unsigned char>(LeftNode,
    RightNode);
00040                     Assert::IsTrue(LeftNode == e->LeftNode());
00041                     Assert::IsTrue(RightNode == e->RightNode());
00042                     delete LeftNode;
00043                     delete RightNode;
00044                     delete e;
00045                 }
00046
00047                 /** @brief test AdjustEdge function
00048                 */
00049                 TEST_METHOD(AdjustEdge) {
00050                     Markov::Node<unsigned char>* LeftNode = new Markov::Node<unsigned char>('l');
00051                     Markov::Node<unsigned char>* RightNode = new Markov::Node<unsigned char>('r');
00052                     Markov::Edge<unsigned char>* e = new Markov::Edge<unsigned char>(LeftNode,
    RightNode);
00053                     e->AdjustEdge(15);
00054                     Assert::AreEqual(15ull, e->EdgeWeight());
00055                     e->AdjustEdge(15);
00056                     Assert::AreEqual(30ull, e->EdgeWeight());
00057                     delete LeftNode;
00058                     delete RightNode;
00059                     delete e;
00060                 }
00061
00062                 /** @brief test TraverseNode returning RightNode
00063                 */
00064                 TEST_METHOD(TraverseNode) {
00065                     Markov::Node<unsigned char>* LeftNode = new Markov::Node<unsigned char>('l');
00066                     Markov::Node<unsigned char>* RightNode = new Markov::Node<unsigned char>('r');
00067                     Markov::Edge<unsigned char>* e = new Markov::Edge<unsigned char>(LeftNode,
    RightNode);
00068                     Assert::IsTrue(RightNode == e->TraverseNode());
00069                     delete LeftNode;
00070                     delete RightNode;
00071                     delete e;
```

```
00072                 }
00073
00074                 /** @brief test LeftNode/RightNode setter
00075                 */
00076                 TEST_METHOD(set_left_and_right) {
00077                     Markov::Node<unsigned char>* LeftNode = new Markov::Node<unsigned char>('l');
00078                     Markov::Node<unsigned char>* RightNode = new Markov::Node<unsigned char>('r');
00079                     Markov::Edge<unsigned char>* e1 = new Markov::Edge<unsigned char>(LeftNode,
    RightNode);
00080
00081                     Markov::Edge<unsigned char>* e2 = new Markov::Edge<unsigned char>;
00082                     e2->SetLeftEdge(LeftNode);
00083                     e2->SetRightEdge(RightNode);
00084
00085                     Assert::IsTrue(e1->LeftNode() == e2->LeftNode());
00086                     Assert::IsTrue(e1->RightNode() == e2->RightNode());
00087                     delete LeftNode;
00088                     delete RightNode;
00089                     delete e1;
00090                     delete e2;
00091                 }
00092
00093                 /** @brief test negative adjustments
00094                 */
00095                 TEST_METHOD(negative_adjust) {
00096                     Markov::Node<unsigned char>* LeftNode = new Markov::Node<unsigned char>('l');
00097                     Markov::Node<unsigned char>* RightNode = new Markov::Node<unsigned char>('r');
00098                     Markov::Edge<unsigned char>* e = new Markov::Edge<unsigned char>(LeftNode,
    RightNode);
00099                     e->AdjustEdge(15);
00100                    Assert::AreEqual(15ull, e->EdgeWeight());
00101                    e->AdjustEdge(-15);
00102                    Assert::AreEqual(0ull, e->EdgeWeight());
00103                    delete LeftNode;
00104                    delete RightNode;
00105                    delete e;
00106                }
00107            };
00108
00109            /** @brief Test class for minimal viable Node
00110            */
00111            TEST_CLASS(Node)
00112            {
00113            public:
00114
00115                /** @brief test default constructor
00116                */
00117                TEST_METHOD(default_constructor) {
00118                    Markov::Node<unsigned char>* n = new Markov::Node<unsigned char>();
00119                    Assert::AreEqual((unsigned char)0, n->NodeValue());
00120                    delete n;
00121                }
00122
00123                /** @brief test custom constructor with unsigned char
00124                */
00125                TEST_METHOD(uchar_constructor) {
00126                    Markov::Node<unsigned char>* n = NULL;
00127                    unsigned char test_cases[] = { 'c', 0x00, 0xff, -32 };
00128                    for (unsigned char tcase : test_cases) {
00129                        n = new Markov::Node<unsigned char>(tcase);
00130                        Assert::AreEqual(tcase, n->NodeValue());
00131                        delete n;
00132                    }
00133                }
00134
00135                /** @brief test link function
00136                */
00137                TEST_METHOD(link_left) {
00138                    Markov::Node<unsigned char>* LeftNode = new Markov::Node<unsigned char>('l');
00139                    Markov::Node<unsigned char>* RightNode = new Markov::Node<unsigned char>('r');
00140
00141                    Markov::Edge<unsigned char>* e = LeftNode->Link(RightNode);
00142                    delete LeftNode;
00143                    delete RightNode;
00144                    delete e;
00145                }
00146
00147                /** @brief test link function
00148                */
00149                TEST_METHOD(link_right) {
00150                    Markov::Node<unsigned char>* LeftNode = new Markov::Node<unsigned char>('l');
00151                    Markov::Node<unsigned char>* RightNode = new Markov::Node<unsigned char>('r');
00152
00153                    Markov::Edge<unsigned char>* e = new Markov::Edge<unsigned char>(NULL, RightNode);
00154                    LeftNode->Link(e);
00155                    Assert::IsTrue(LeftNode == e->LeftNode());
00156                    Assert::IsTrue(RightNode == e->RightNode());
```

```
00157                     delete LeftNode;
00158                     delete RightNode;
00159                     delete e;
00160                 }
00161
00162                 /** @brief test RandomNext with low values
00163                 */
00164                 TEST_METHOD(rand_next_low) {
00165                     Markov::Random::Marsaglia MarsagliaRandomEngine;
00166                     Markov::Node<unsigned char>* src = new Markov::Node<unsigned char>('a');
00167                     Markov::Node<unsigned char>* target1 = new Markov::Node<unsigned char>('b');
00168                     Markov::Edge<unsigned char>* e = src->Link(target1);
00169                     e->AdjustEdge(15);
00170                     Markov::Node<unsigned char>* res = src->RandomNext(&MarsagliaRandomEngine);
00171                     Assert::IsTrue(res == target1);
00172                     delete src;
00173                     delete target1;
00174                     delete e;
00175
00176                 }
00177
00178                 /** @brief test RandomNext with 32 bit high values
00179                 */
00180                 TEST_METHOD(rand_next_u32) {
00181                     Markov::Random::Marsaglia MarsagliaRandomEngine;
00182                     Markov::Node<unsigned char>* src = new Markov::Node<unsigned char>('a');
00183                     Markov::Node<unsigned char>* target1 = new Markov::Node<unsigned char>('b');
00184                     Markov::Edge<unsigned char>* e = src->Link(target1);
00185                     e->AdjustEdge(1 « 31);
00186                     Markov::Node<unsigned char>* res = src->RandomNext(&MarsagliaRandomEngine);
00187                     Assert::IsTrue(res == target1);
00188                     delete src;
00189                     delete target1;
00190                     delete e;
00191
00192                 }
00193
00194                 /** @brief random next on a node with no follow-ups
00195                 */
00196                 TEST_METHOD(rand_next_choice_1) {
00197                     Markov::Random::Marsaglia MarsagliaRandomEngine;
00198                     Markov::Node<unsigned char>* src = new Markov::Node<unsigned char>('a');
00199                     Markov::Node<unsigned char>* target1 = new Markov::Node<unsigned char>('b');
00200                     Markov::Node<unsigned char>* target2 = new Markov::Node<unsigned char>('c');
00201                     Markov::Edge<unsigned char>* e1 = src->Link(target1);
00202                     Markov::Edge<unsigned char>* e2 = src->Link(target2);
00203                     e1->AdjustEdge(1);
00204                     e2->AdjustEdge((unsigned long)(1ull « 31));
00205                     Markov::Node<unsigned char>* res = src->RandomNext(&MarsagliaRandomEngine);
00206                     Assert::IsNotNull(res);
00207                     Assert::IsTrue(res == target2);
00208                     delete src;
00209                     delete target1;
00210                     delete e1;
00211                     delete e2;
00212                 }
00213
00214                 /** @brief random next on a node with no follow-ups
00215                 */
00216                 TEST_METHOD(rand_next_choice_2) {
00217                     Markov::Random::Marsaglia MarsagliaRandomEngine;
00218                     Markov::Node<unsigned char>* src = new Markov::Node<unsigned char>('a');
00219                     Markov::Node<unsigned char>* target1 = new Markov::Node<unsigned char>('b');
00220                     Markov::Node<unsigned char>* target2 = new Markov::Node<unsigned char>('c');
00221                     Markov::Edge<unsigned char>* e1 = src->Link(target1);
00222                     Markov::Edge<unsigned char>* e2 = src->Link(target2);
00223                     e2->AdjustEdge(1);
00224                     e1->AdjustEdge((unsigned long)(1ull « 31));
00225                     Markov::Node<unsigned char>* res = src->RandomNext(&MarsagliaRandomEngine);
00226                     Assert::IsNotNull(res);
00227                     Assert::IsTrue(res == target1);
00228                     delete src;
00229                     delete target1;
00230                     delete e1;
00231                     delete e2;
00232                 }
00233
00234
00235                 /** @brief test updateEdges
00236                 */
00237                 TEST_METHOD(update_edges_count) {
00238
00239                     Markov::Node<unsigned char>* src = new Markov::Node<unsigned char>('a');
00240                     Markov::Node<unsigned char>* target1 = new Markov::Node<unsigned char>('b');
00241                     Markov::Node<unsigned char>* target2 = new Markov::Node<unsigned char>('c');
00242                     Markov::Edge<unsigned char>* e1 = new Markov::Edge<unsigned char>(src, target1);
00243                     Markov::Edge<unsigned char>* e2 = new Markov::Edge<unsigned char>(src, target2);
```

```
00244                   e1->AdjustEdge(25);
00245                   src->UpdateEdges(e1);
00246                   e2->AdjustEdge(30);
00247                   src->UpdateEdges(e2);
00248
00249                   Assert::AreEqual((size_t)2, src->Edges()->size());
00250
00251                   delete src;
00252                   delete target1;
00253                   delete e1;
00254                   delete e2;
00255
00256               }
00257
00258               /** @brief test updateEdges
00259               */
00260               TEST_METHOD(update_edges_total) {
00261
00262                   Markov::Node<unsigned char>* src = new Markov::Node<unsigned char>('a');
00263                   Markov::Node<unsigned char>* target1 = new Markov::Node<unsigned char>('b');
00264                   Markov::Edge<unsigned char>* e1 = new Markov::Edge<unsigned char>(src, target1);
00265                   Markov::Edge<unsigned char>* e2 = new Markov::Edge<unsigned char>(src, target1);
00266                   e1->AdjustEdge(25);
00267                   src->UpdateEdges(e1);
00268                   e2->AdjustEdge(30);
00269                   src->UpdateEdges(e2);
00270
00271                   //Assert::AreEqual(55ull, src->TotalEdgeWeights());
00272
00273                   delete src;
00274                   delete target1;
00275                   delete e1;
00276                   delete e2;
00277               }
00278
00279
00280               /** @brief test FindVertice
00281               */
00282               TEST_METHOD(find_vertice) {
00283
00284                   Markov::Node<unsigned char>* src = new Markov::Node<unsigned char>('a');
00285                   Markov::Node<unsigned char>* target1 = new Markov::Node<unsigned char>('b');
00286                   Markov::Node<unsigned char>* target2 = new Markov::Node<unsigned char>('c');
00287                   Markov::Edge<unsigned char>* res = NULL;
00288                   src->Link(target1);
00289                   src->Link(target2);
00290
00291
00292                   res = src->FindEdge('b');
00293                   Assert::IsNotNull(res);
00294                   Assert::AreEqual((unsigned char)'b', res->TraverseNode()->NodeValue());
00295                   res = src->FindEdge('c');
00296                   Assert::IsNotNull(res);
00297                   Assert::AreEqual((unsigned char)'c', res->TraverseNode()->NodeValue());
00298
00299                   delete src;
00300                   delete target1;
00301                   delete target2;
00302
00303
00304               }
00305
00306
00307               /** @brief test FindVertice
00308               */
00309               TEST_METHOD(find_vertice_without_any) {
00310
00311                   auto _invalid_next = [] {
00312                       Markov::Node<unsigned char>* src = new Markov::Node<unsigned char>('a');
00313                       Markov::Edge<unsigned char>* res = NULL;
00314
00315                       res = src->FindEdge('b');
00316                       Assert::IsNull(res);
00317
00318                       delete src;
00319                   };
00320
00321                   //Assert::ExpectException<std::logic_error>(_invalid_next);
00322               }
00323
00324               /** @brief test FindVertice
00325               */
00326               TEST_METHOD(find_vertice_nonexistent) {
00327
00328                   Markov::Node<unsigned char>* src = new Markov::Node<unsigned char>('a');
00329                   Markov::Node<unsigned char>* target1 = new Markov::Node<unsigned char>('b');
00330                   Markov::Node<unsigned char>* target2 = new Markov::Node<unsigned char>('c');
```

```
00331                   Markov::Edge<unsigned char>* res = NULL;
00332                   src->Link(target1);
00333                   src->Link(target2);
00334
00335                   res = src->FindEdge('D');
00336                   Assert::IsNull(res);
00337
00338                   delete src;
00339                   delete target1;
00340                   delete target2;
00341
00342               }
00343           };
00344
00345           /** @brief Test class for minimal viable Model
00346           */
00347           TEST_CLASS(Model)
00348           {
00349           public:
00350               /** @brief test model constructor for starter node
00351               */
00352               TEST_METHOD(model_constructor) {
00353                   Markov::Model<unsigned char> m;
00354                   Assert::AreEqual((unsigned char)'\0', m.StarterNode()->NodeValue());
00355               }
00356
00357               /** @brief test import
00358               */
00359               TEST_METHOD(import_filename) {
00360                   Markov::Model<unsigned char> m;
00361                   Assert::IsTrue(m.Import("../MarkovPasswords/Models/2gram.mdl"));
00362               }
00363
00364               /** @brief test export
00365               */
00366               TEST_METHOD(export_filename) {
00367                   Markov::Model<unsigned char> m;
00368                   Assert::IsTrue(m.Export("../MarkovPasswords/Models/testcase.mdl"));
00369               }
00370
00371               /** @brief test random walk
00372               */
00373               TEST_METHOD(random_walk) {
00374                   unsigned char* res = new unsigned char[12 + 5];
00375                   Markov::Random::Marsaglia MarsagliaRandomEngine;
00376                   Markov::Model<unsigned char> m;
00377                   Assert::IsTrue(m.Import("../Models/finished2.mdl"));
00378                   Assert::IsNotNull(m.RandomWalk(&MarsagliaRandomEngine,1,12,res));
00379               }
00380           };
00381       }
00382
00383       /** @brief Testing namespace for MVP MarkovPasswords
00384       */
00385       namespace MarkovPasswords
00386       {
00387           /** @brief Test Class for Argparse class
00388           */
00389           TEST_CLASS(ArgParser)
00390           {
00391           public:
00392               /** @brief test basic generate
00393               */
00394               TEST_METHOD(generate_basic) {
00395                   int argc = 8;
00396                   char *argv[] = {"markov.exe", "generate", "-if", "model.mdl", "-of",
      "passwords.txt", "-n", "100"};
00397
00398                   /*ProgramOptions *p = Argparse::parse(argc, argv);
00399                   Assert::IsNotNull(p);
00400
00401                   Assert::AreEqual(p->bImport, true);
00402                   Assert::AreEqual(p->bExport, false);
00403                   Assert::AreEqual(p->importname, "model.mdl");
00404                   Assert::AreEqual(p->outputfilename, "passwords.txt");
00405                   Assert::AreEqual(p->generateN, 100); */
00406
00407               }
00408
00409               /** @brief test basic generate reordered params
00410               */
00411               TEST_METHOD(generate_basic_reorder) {
00412                   int argc = 8;
00413                   char *argv[] = { "markov.exe", "generate", "-n", "100", "-if", "model.mdl", "-of",
      "passwords.txt" };
00414
00415                   /*ProgramOptions* p = Argparse::parse(argc, argv);
```

```
00416                   Assert::IsNotNull(p);
00417
00418                   Assert::AreEqual(p->bImport, true);
00419                   Assert::AreEqual(p->bExport, false);
00420                   Assert::AreEqual(p->importname, "model.mdl");
00421                   Assert::AreEqual(p->outputfilename, "passwords.txt");
00422                   Assert::AreEqual(p->generateN, 100);*/
00423               }
00424
00425               /** @brief test basic generate param longnames
00426               */
00427               TEST_METHOD(generate_basic_longname) {
00428                   int argc = 8;
00429                   char *argv[] = { "markov.exe", "generate", "-n", "100", "--inputfilename",
    "model.mdl", "--outputfilename", "passwords.txt" };
00430
00431                   /*ProgramOptions* p = Argparse::parse(argc, argv);
00432                   Assert::IsNotNull(p);
00433
00434                   Assert::AreEqual(p->bImport, true);
00435                   Assert::AreEqual(p->bExport, false);
00436                   Assert::AreEqual(p->importname, "model.mdl");
00437                   Assert::AreEqual(p->outputfilename, "passwords.txt");
00438                   Assert::AreEqual(p->generateN, 100); */
00439               }
00440
00441               /** @brief test basic generate
00442               */
00443               TEST_METHOD(generate_fail_badmethod) {
00444                   int argc = 8;
00445                   char *argv[] = { "markov.exe", "junk", "-n", "100", "--inputfilename",
    "model.mdl", "--outputfilename", "passwords.txt" };
00446
00447                   /*ProgramOptions* p = Argparse::parse(argc, argv);
00448                   Assert::IsNull(p); */
00449               }
00450
00451               /** @brief test basic train
00452               */
00453               TEST_METHOD(train_basic) {
00454                   int argc = 4;
00455                   char *argv[] = { "markov.exe", "train", "-ef", "model.mdl" };
00456
00457                   /*ProgramOptions* p = Argparse::parse(argc, argv);
00458                   Assert::IsNotNull(p);
00459
00460                   Assert::AreEqual(p->bImport, false);
00461                   Assert::AreEqual(p->bExport, true);
00462                   Assert::AreEqual(p->exportname, "model.mdl"); */
00463
00464               }
00465
00466               /** @brief test basic generate
00467               */
00468               TEST_METHOD(train_basic_longname) {
00469                   int argc = 4;
00470                   char *argv[] = { "markov.exe", "train", "--exportfilename", "model.mdl" };
00471
00472                   /*ProgramOptions* p = Argparse::parse(argc, argv);
00473                   Assert::IsNotNull(p);
00474
00475                   Assert::AreEqual(p->bImport, false);
00476                   Assert::AreEqual(p->bExport, true);
00477                   Assert::AreEqual(p->exportname, "model.mdl"); */
00478               }
00479
00480
00481
00482           };
00483
00484       }
00485   }
00486
00487
00488   /** @brief Testing namespace for MarkovModel
00489   */
00490   namespace MarkovModel {
00491
00492       /** @brief Test class for rest of Edge cases
00493       */
00494       TEST_CLASS(Edge)
00495       {
00496       public:
00497           /** @brief send exception on integer underflow
00498           */
00499           TEST_METHOD(except_integer_underflow) {
00500               auto _underflow_adjust = [] {
```

```
00501                   Markov::Node<unsigned char>* LeftNode = new Markov::Node<unsigned char>('l');
00502                   Markov::Node<unsigned char>* RightNode = new Markov::Node<unsigned char>('r');
00503                   Markov::Edge<unsigned char>* e = new Markov::Edge<unsigned char>(LeftNode,
      RightNode);
00504                   e->AdjustEdge(15);
00505                   e->AdjustEdge(-30);
00506                   delete LeftNode;
00507                   delete RightNode;
00508                   delete e;
00509               };
00510               Assert::ExpectException<std::underflow_error>(_underflow_adjust);
00511           }
00512
00513           /** @brief test integer overflows
00514           */
00515           TEST_METHOD(except_integer_overflow) {
00516               auto _overflow_adjust = [] {
00517                   Markov::Node<unsigned char>* LeftNode = new Markov::Node<unsigned char>('l');
00518                   Markov::Node<unsigned char>* RightNode = new Markov::Node<unsigned char>('r');
00519                   Markov::Edge<unsigned char>* e = new Markov::Edge<unsigned char>(LeftNode,
      RightNode);
00520                   e->AdjustEdge(~0ull);
00521                   e->AdjustEdge(1);
00522                   delete LeftNode;
00523                   delete RightNode;
00524                   delete e;
00525               };
00526               Assert::ExpectException<std::underflow_error>(_overflow_adjust);
00527           }
00528       };
00529
00530       /** @brief Test class for rest of Node cases
00531       */
00532       TEST_CLASS(Node)
00533       {
00534       public:
00535
00536           /** @brief test RandomNext with 64 bit high values
00537           */
00538           TEST_METHOD(rand_next_u64) {
00539               Markov::Random::Marsaglia MarsagliaRandomEngine;
00540               Markov::Node<unsigned char>* src = new Markov::Node<unsigned char>('a');
00541               Markov::Node<unsigned char>* target1 = new Markov::Node<unsigned char>('b');
00542               Markov::Edge<unsigned char>* e = src->Link(target1);
00543               e->AdjustEdge((unsigned long)(1ull << 63));
00544               Markov::Node<unsigned char>* res = src->RandomNext(&MarsagliaRandomEngine);
00545               Assert::IsTrue(res == target1);
00546               delete src;
00547               delete target1;
00548               delete e;
00549
00550           }
00551
00552           /** @brief test RandomNext with 64 bit high values
00553           */
00554           TEST_METHOD(rand_next_u64_max) {
00555               Markov::Random::Marsaglia MarsagliaRandomEngine;
00556               Markov::Node<unsigned char>* src = new Markov::Node<unsigned char>('a');
00557               Markov::Node<unsigned char>* target1 = new Markov::Node<unsigned char>('b');
00558               Markov::Edge<unsigned char>* e = src->Link(target1);
00559               e->AdjustEdge((0xffffFFFF));
00560               Markov::Node<unsigned char>* res = src->RandomNext(&MarsagliaRandomEngine);
00561               Assert::IsTrue(res == target1);
00562               delete src;
00563               delete target1;
00564               delete e;
00565
00566           }
00567
00568           /** @brief randomNext when no edges are present
00569           */
00570           TEST_METHOD(uninitialized_rand_next) {
00571
00572               auto _invalid_next = [] {
00573                   Markov::Random::Marsaglia MarsagliaRandomEngine;
00574                   Markov::Node<unsigned char>* src = new Markov::Node<unsigned char>('a');
00575                   Markov::Node<unsigned char>* target1 = new Markov::Node<unsigned char>('b');
00576                   Markov::Edge<unsigned char>* e = new Markov::Edge<unsigned char>(src, target1);
00577                   Markov::Node<unsigned char>* res = src->RandomNext(&MarsagliaRandomEngine);
00578
00579                   delete src;
00580                   delete target1;
00581                   delete e;
00582               };
00583
00584               Assert::ExpectException<std::logic_error>(_invalid_next);
00585           }
```

```
00586
00587
00588         };
00589
00590         /** @brief Test class for rest of model cases
00591         */
00592         TEST_CLASS(Model)
00593         {
00594         public:
00595             TEST_METHOD(functional_random_walk) {
00596                 unsigned char* res2 = new unsigned char[12 + 5];
00597                 Markov::Random::Marsaglia MarsagliaRandomEngine;
00598                 Markov::Model<unsigned char> m;
00599                 Markov::Node<unsigned char>* starter = m.StarterNode();
00600                 Markov::Node<unsigned char>* a = new Markov::Node<unsigned char>('a');
00601                 Markov::Node<unsigned char>* b = new Markov::Node<unsigned char>('b');
00602                 Markov::Node<unsigned char>* c = new Markov::Node<unsigned char>('c');
00603                 Markov::Node<unsigned char>* end = new Markov::Node<unsigned char>(0xff);
00604                 starter->Link(a)->AdjustEdge(1);
00605                 a->Link(b)->AdjustEdge(1);
00606                 b->Link(c)->AdjustEdge(1);
00607                 c->Link(end)->AdjustEdge(1);
00608
00609                 char* res = (char*)m.RandomWalk(&MarsagliaRandomEngine,1,12,res2);
00610                 Assert::IsFalse(strcmp(res, "abc"));
00611             }
00612             TEST_METHOD(functionoal_random_walk_without_any) {
00613                 Markov::Model<unsigned char> m;
00614                 Markov::Node<unsigned char>* starter = m.StarterNode();
00615                 Markov::Node<unsigned char>* a = new Markov::Node<unsigned char>('a');
00616                 Markov::Node<unsigned char>* b = new Markov::Node<unsigned char>('b');
00617                 Markov::Node<unsigned char>* c = new Markov::Node<unsigned char>('c');
00618                 Markov::Node<unsigned char>* end = new Markov::Node<unsigned char>(0xff);
00619                 Markov::Edge<unsigned char>* res = NULL;
00620                 starter->Link(a)->AdjustEdge(1);
00621                 a->Link(b)->AdjustEdge(1);
00622                 b->Link(c)->AdjustEdge(1);
00623                 c->Link(end)->AdjustEdge(1);
00624
00625                 res = starter->FindEdge('D');
00626                 Assert::IsNull(res);
00627
00628             }
00629         };
00630
00631     }
00632
00633     /** @brief Testing namespace for MarkovPasswords
00634     */
00635     namespace MarkovPasswords {
00636
00637     };
00638
00639 }
```

Index