Genealogical Tree

Find all the descendant on any level of ancestry

0.0.1_fea9b06

Generated:
Fri Jul 3 2015 21:53:47 on user@ubuntu

Git Commit Hash:
fea9b06

Git Details:
fea9b06 (HEAD, origin/develop, origin/HEAD, develop) Regenerated PDF without VERSION-HARDCODED option

Contents

Chapter 1

Genealogical Tree

Summary

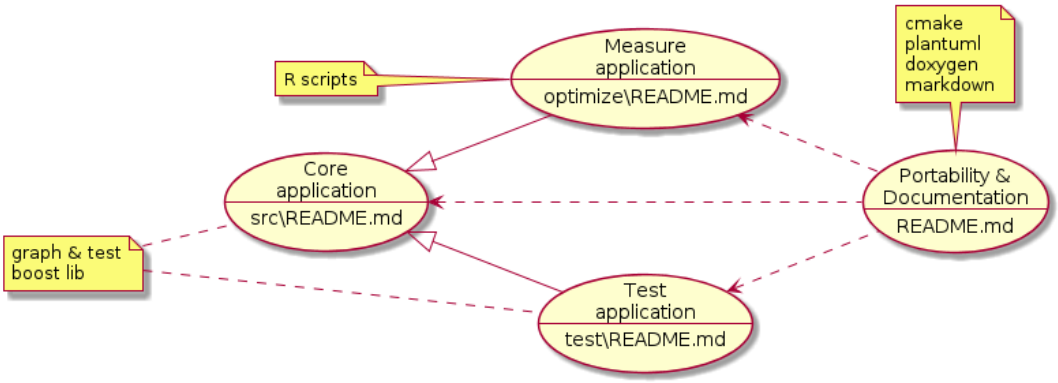
Program should be able to **find all the descendant with name Bob for all the ascendants with name Will on any level of ancestry**. In order to present the capabilities of your app:

- implement the application to optimize the initialization time.
- application should have built in data about genealogical tree of people living in particular country.
- please generate a representative data that has sample people an relationships between them. Use all varieties of names (can be also generated) but also put two test names (Bob and Will) and connect them in different relationships.
- the application should posses tests that are checking possible edge cases and ensure the stability of the application.
- the designed data structure should ensure optimized search time on following fields: name, last name, date of birth and location.

Approach

Instead of starting directly with the problem core, don't test thoroughly edge cases, leaping into too early optimization, don't document your results/decisions/mistakes and ending with an app that only run partially on your development environment, the **aproach** will be the opposite one.

1. Ensure a minimum of portability on different environments.
2. Document as much automatically as possible to draw conclusions from your mistakes and let others reproduce your results.
3. Write meaningful tests to cover your app and let you tackle optimizations knowing you're not breaking previous development.
4. Measure your application in order to compare improvements/regressions during the optimization stage.
5. Solve the core problem in the most simple and maintainable way at our disposal.



No doubt this approach is an overkill for a pet project but it's way more realistic for big, long C++ ones.

Portability and documentation

A Modern C++ GNU compiler, g++ 4.9.2 or above, and a recent cmake, 3.1 or above, are the minimum. As well a valid *boost* library is supposed to be installed.

Regarding to documentation, *doxygen*, *latex*, *graphviz* and *plantuml.jar* are needed. For example, if you work with Xubuntu 15.04 or its **Docker** equivalent, the following commands might do the trick for you:

```
sudo apt-get -y install git build-essential libboost-all-dev
sudo apt-get -y install doxygen doxygen-latex openjdk-8-jdk graphviz
sudo add-apt-repository -y ppa:george-edison55/cmake-3.x
sudo apt-get -y update
sudo apt-get -y install cmake
sudo apt-get -y upgrade
sudo mkdir /opt/plantuml && sudo chmod a+wr /opt/plantuml
wget http://sourceforge.net/projects/plantuml/files/plantuml.jar/download \
     -O /opt/plantuml/plantuml.jar
```

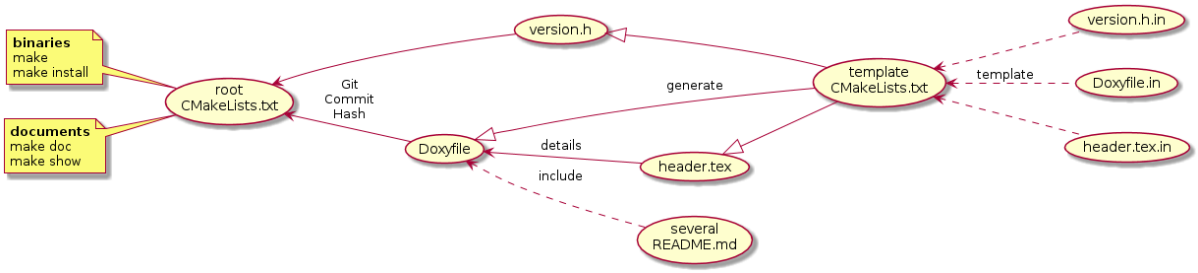
For other O.S., have a look to Homebrew or Git/MinGW

Generate binaries & documentation

Usual commands:

```
mkdir build
cd build
cmake ..
make
make doc
```

Optionally you can invoke *make install* to install binaries or *make show* to install documentation utility.



Note: If you happen to work with *OSX* and Homebrew, don't forget to invoke *cmake* pointing to the **GNU** compiler:

```
cmake -DCMAKE_CXX_COMPILER=/usr/local/bin/g++-5 ..
```

Note: If you happen to work with *Windows* and Git/MinGW, don't forget to invoke *cmake* pointing to the **GNU** generator:

```
cmake -G "MSYS Makefiles" ..
```

As well a script, called **show** or something similar, will be created in your *home* directory as a shortcut for generating & viewing documentation. Don't hesitate to use it as a *template* for your specific environment.

Generate only documentation

Similar commands to the previous ones, just the compiler is not required:

```
mkdir build
cd build
cmake -DONLY_DOC=TRUE ..
make doc
```

Note: If you happen to work with *Windows* and Git/MinGW, don't forget to invoke *cmake* pointing to the **GNU** generator:

```
cmake -G "MSYS Makefiles" -DONLY_DOC=TRUE ..
```

Note: If your **make** utility is not installed in the default place, define *CMAKE_BUILD_TOOL*

```
cmake -G "MSYS Makefiles" -DCMAKE_BUILD_TOOL=<your location> -DONLY_DOC=TRUE ..
```

As well, if you installed the documentation utility with **make show**, you're supposed to able to recreate and view that documentation PDF though usual *ssh* connection with enabled X11:

```
ssh -X <user>@<location> "./show"
```

Development details

In order to generate binaries & documentation, the following versions were used:

For code

Pay attention to *cmake* and *gcc* versions. A minimum is required to work on several O.S. using modern C++. Feel free to locally hack **CMakeLists.txt** to meet your needs.

Linux (Xubuntu 15.04)

- **cmake** *3.2.2*
- **gcc** *4.9.2*
- **boost** *1.55*

OSX (Yosemite 10.10.4)

- **cmake** *3.2.2*
- **gcc** *5.1*
- **boost** *1.58*

Note: If you happen to work with *OSX* and Homebrew, don't forget to compile **boost** with the previous **gcc** compiler, not with the default *clang* one:

```
brew install gcc
brew install boost --cc=gcc-5
```

Windows (Win7 x64)

- **cmake** *3.3.0*
- **gcc** *5.1*
- **boost** *1.58*

For documentation

Environment variables to locate PlantUML *jar* and default *PDF* viewer can be defined to overwrite default values. See **CMakeLists.txt** for further information on your platform.

Linux

- **doxygen** *1.8.9.1*
- **latex**/**pdfTeX** *2.6-1.40.15*
- **graphviz**/**dot** *2.38.0*
- **java**/**plantuml** *1.8.0_45/8026*

OSX

- **doxygen** *1.8.9.1*
- **latex**/**pdfTeX** *2.6-1.40.15*
- **graphviz**/**dot** *2.38.0*
- **java**/**plantuml** *1.8.0_40/8026*

Windows

- **doxygen** *1.8.9.1*
- **latex**/**pdfTeX** *2.9.5496-1.40.15*
- **graphviz**/**dot** *2.38.0*
- **java**/**plantuml** *1.8.0_45/8026*

Note: Don't forget configure *Doxyfile* and *CMakeLists.txt* to use **README.md** as *Main Page* for **latex** documentation.

For IDE

To use **NetBeans** don't forget to configure a *cmake* project with *custom* **build** folder. Add at that moment any extra customization in the command line used by *cmake* instruction. For example:

- -DCMAKE_CXX_COMPILER=g++-5 for **OSX**
- -DONLY_DOC=TRUE for only documentation on **Linux**/**OSX**
- -G "MSYS Makefiles" for **Windows**
- -G "MSYS Makefiles" -DONLY_DOC=TRUE for only documentation on **Windows**

Note: If you happen to use *jVi* plugin on *OSX*, don't forget to use "-lc" instead of just "-c" for its /bin/bash flag.

GIT Commit Hash

In order to add the specific **git commit hash** into code & documentation, *templates* are defined in the *template* folder for **Doxyfile**, **header.tex** & **version.h** files.

In order to **speed up** local compilations and let us hardcode our locally generated files, it's possible to instruct *cmake* to use this hardcoded header instead of usual GIT one.

The parameter to pass onto **cmake** is **VERSION_HARDCODED**:

```
cmake <rest of options> -DVERSION_HARDCODED=TRUE ..
```

Chapter 2

Test your application

Taking advantage of *boost* test cases as explained at An Engineer's Guide to Unit Testing.

Chapter 3

Measure your application

Scripts to gather information on performance. Basically *statistical information* on execution time of different approaches.

Chapter 4

Application Core Code

Source folder for headears & code files directly involved with the core problem.

Limits

First of all, we should grasp a rough idea about which range of numbers to consider:

- Most Populous Country: China
  Inhabitants: around 1400000000
- Another populous country, culturally diverse: USA
  Number of first & last names: around 5200 & 152000
- Example of baptism registers: Ireland
  Roman Catholic: around 19th century
- Marriageable age: world
  Some common value: around 20
- Number of locations: India
  Number of villages: around 640000

Assumptions on numbers

This way we can assume that taking into account around 200 years of sensible information on our ascendants, around 10 generations back in time, we suppose not to deal with more than 4000000000 individuals.

As well, we could consider that our application should only tackle around different 6000 first names or 60000 last names in our given country. Even we can take for granted that there aren't more than 60000 locations, that we might classify them in two levels; one coarse level easy to remember and another fine one more close to small places.

Translate into C++:

- First Name: unsigned short int (uint16_t)
- Last Name: unsigned short int (uint16_t)
- Year of Birth: unsigned char (unit8_t) $<$ 200 years
- Coarse Location of Birth: unsigned short int (unit8_t)
- Month of Birth: unsigned char (uint8_t)
- Day of Birth: unsigned char (unit8_t)

- Fine location of Birth: unsigned short int (uint8_t)
- More information related to a specific subject: extra indexes.

This way we can use the **first 64 bits of information** as a valid **identification** for the individuals and with the advantage of getting the relevant information to debug first: *name and generation*.

Generated Files

version.h is generated with *GIT* information by *cmake*.

But in order to **speed up** local compilations and let us hardcode our locally generated files, it's possible to instruct *cmake* to use this hardcoded header instead of usual GIT one.

The parameter to pass onto **cmake** is **VERSION_HARDCODED**:

```
cmake <rest of options> -DVERSION_HARDCODED=TRUE ..
```

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 documented files with brief descriptions:

Chapter 7

Class Documentation

7.1 struct_index_t Struct Reference

Public Attributes

- char **year**

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

- src/id.h

7.2 union_id_t Union Reference

Public Attributes

- uint64_t **id**
- struct_index_t **index**

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

- src/id.h

Chapter 8

File Documentation

8.1 src/id.h File Reference

```
#include <cstdint>
#include <iostream>
```

Classes

- struct struct_index_t
- union union_id_t

Functions

- std::ostream & **operator**<< (std::ostream &os, const union_id_t &u)

Variables

- static constexpr const union_id_t **EMPTY_UNION_ID** {0}

8.1.1 Detailed Description

Define types for id's for our subjects

A first approach of getting packed id & basic information in form of indexes:

- First Name: unsigned short int (uint16_t)
- Last Name: unsigned short int (uint16_t)
- Year of Birth: unsigned char (unit8_t) < 200 years
- Coarse Location of Birth: unsigned short int (unit8_t)
- Month of Birth: unsigned char (uint8_t)
- Day of Birth: unsigned char (unit8_t)

Grouping all that indexes we got a 64 bits identification

8.2 src/main.cpp File Reference

```
#include <iostream>
#include "version.h"
#include "id.h"
```

Functions

- int main (int argc, char **argv)
  Main function.

8.2.1 Function Documentation

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

Main function.

Parameters

argc	An integer argument count of the command line arguments
argv	An argument vector of the command line arguments

Returns

an integer 0 upon exit success

8.3 src/version.h File Reference

Macros

- #define **DEFINE_VERSION_FIRST** "0"
- #define **DEFINE_VERSION_MIDDLE** "0"
- #define **DEFINE_VERSION_LAST** "1"
- #define **DEFINE_GIT_DETAILS** "fea9b06 (HEAD, origin/develop, origin/HEAD, develop) Regenerated PDF without VERSION-HARDCODED option"
- #define **DEFINE_GIT_COMMIT_HASH** "fea9b06"
- #define **DEFINE_VERSION**

Variables

- static const char * **VERSION** = "VERSION = " DEFINE_VERSION
- static const char * **GIT_DETAILS** = "GIT_DETAILS = " DEFINE_GIT_DETAILS

8.3.1 Detailed Description

This metadata information might be located through **strings** command

- Linux/Solaris/Mac:

```
strings <binary> | grep VERSION
strings <binary> | grep GIT_DETAILS
```

- Windows (MinGW):

```
strings <binary> | findstr VERSION
strings <binary> | findstr GIT_DETAILS
```

8.3.2 Macro Definition Documentation

8.3.2.1 #define DEFINE_VERSION

Value:

```
DEFINE_VERSION_FIRST "." \
        DEFINE_VERSION_MIDDLE "." DEFINE_VERSION_LAST \
    "_" DEFINE_GIT_COMMIT_HASH
```

Index