****

**JIANGSU UNIVERSITY**

数据结构课程设计

课程名称： 专业班级：

学生学号： 学生姓名：

指导教师： 实验日期：

**2020——2021学年 第 一 学期**

# 一、目录

# 二、设计目标

通过本课程的课程设计，应使学生达到以下的基本要求:

1. 能根据加工数据对象的特征，选择适当的数据结构、存贮结构及相应算法，初步掌握各种算法在时间和空间的分析技巧。

2. 能够进行算法设计和程序设计，并且使所设计的程序结构清楚，正确易读，并上机调试通过。

3. 较强的实习、实践能力。

4. 较强的分析问题、解决问题的能力。

# 三、问题描述

1．题目

家谱管理系统。

2．要求

（1）输入文件以存放最初家谱中各成员的信息，成员的信息中均应包含以下内容：姓名、出生日期、婚否、地址、健在否、死亡日期（若其已死亡），也可附加其它信息、但不是必需的。

（2）实现数据的文件存储和读取。

（3）以图形方式显示家谱。

（4）显示第n 代所有人的信息。

（5）按照姓名查询，输出成员信息（包括其本人、父亲、孩子的信息）。

（6）按照出生日期查询成员名单。

（7）输入两人姓名，确定其关系。

（8）某成员添加孩子。

（9）删除某成员（若其还有后代，则一并删除）。

（10）修改某成员信息。

# 四、系统分析与概要设计

1．需求分析

本家谱的功能有打印家谱，根据名词、第几代、生日等条件查找成员，删除成员，添加成员，修改成员信息的功能，同时需要支持从文件中读取数据以及将数据存储到文件中。

2. 设计思路

首先抽象出家族成员的实体模型，采用合适的数据结构进行存储和表示。然后针对具体的功能需求设计相对友好的用户界面，对用户输入具有较好的容错能力，根据设计好的原型定义所需的方法和操作以及必须的文件等，最终完成编码和调试工作。

3. 数据结构的设计

本家谱管理系统使用多叉树的结构来实现，每个节点为Person对象，父节点为父亲，子节点为孩子。

class Person

{

private:

string name;/\*姓名\*/

string birthday;/\*出生日期\*/

bool married;/\*是否婚配\*/

string address;/\*地址\*/

bool live;/\*是否健在\*/

string deathDate;/\*死亡日期\*/

int id;/\*唯一标识\*/

vector<Person\*> children; //子女的指针容器

Person\* father; //指向父亲的指针

public:

Person();

~Person();

//各私有成员的get、set方法。此处略

/\*获取孩子列表引用\*/

vector<Person\*>& getChildren();

/\*设置孩子列表\*/

void setChildren(vector<Person\*> children);

/\*获取父亲指针\*/

Person\* getFather();

/\*设置父亲之中\*/

void setFather(Person\* father);

/\*添加孩子\*/

void addChild(Person\* child);

/\*显示成员信息\*/

void display();

};

4．概要设计

（1）Person类

定义了本系统中的基本数据结构，包含家族成员的信息以及所需的操作。

（2）GMS类

Genealogy Management System的简写。这是直接与用户进行交互的模块类。其基本功能如下：

/\*家谱管理系统类\*/

class GMS

{

private:

/\*家族祖先\*/

Person\* ancestor;

/\*显示菜单\*/

void showMenu();

/\*显示家谱\*/

void displayGenealogy();

/\*按层级展示家谱\*/

void showLevel();

/\*展示所有成员列表\*/

void showAllPersons(Person\* root);

/\*显示第n代所有成员\*/

void displayGeneration();

/\*根据代数查找\*/

void searchByGeneration(Person\* root, int tarGen, int curGen, vector<Person\*>& persons);

/\*根据名称查找\*/

void searchByName();

Person\* searchByName(Person\* root, string name);

/\*根据生日查找\*/

void searchByBirthday();

void searchByBirthday(Person\* root, string birthday, vector<Person\*>& persons);

/\*输入两人姓名，判断两人关系\*/

void getRelationshipByNames();

/\*添加孩子\*/

void addChildren();

/\*删除成员\*/

void remove();

void remove(Person\* root);

/\*修改成员\*/

void update();

/\*退出程序并写入文件\*/

void quit();

/\*显示成员列表表头\*/

void showTableHeader();

/\*设置祖先\*/

void setAncestor();

/\*修改指定成员\*/

void modifyPerson(Person\* p);

/\*家谱文件路径\*/

static const string filePath;

/\*读取文件加载家谱信息\*/

void readFile();

/\*将家谱信息写入文件\*/

void writeFile();

void writeFile(Person\* root, ofstream& fout);

public:

GMS();

~GMS();

/\*运行家谱管理系统\*/

void start();

};

（3）IdGenerator类

此为成员ID生成的工具类。

/\*唯一标识生成器\*/

class IdGenerator

{

private:

static int current;/\*当前最新id值 + 1\*/

public:

/\*生成id\*/

int static generate();

/\*修改当前最新id值\*/

void static update(int id);

};

# 五、详细设计

1. 关键算法流程

（1）文件读写

读取文件：打开文件输入流，读取表头后逐行读取并创建Person对象，根据文件中所记录的父亲姓名建立节点间链接关系，最后关闭文件。

写入文件：打开文件输出流，写入表头后先序遍历家族树，每个家族成员为1行，写入完毕后关闭文件。

（2）家谱展示

首先使用树结构的层序遍历展示节点间层次关系，再用表格形式将所有成员的详细信息展示出来。

（3）按代数查询

采用先序遍历方式当层数为目标值时添加至容器中，最后将容器中得到的成员依次输出。

（4）按姓名查询

采用先序遍历方式查找指定姓名的成员并返回，若查找失败返回NULL

（5）按出生日期查询

采用先序遍历方式当出生日期与参数相等时添加至容器中，最后将容器中得到的成员依次输出。

（6）输入两人姓名，确定其关系

查找得到两个成员p1, p2，若有一个为空则报错提醒。分别计算p1，p2的代数，得出代差。根据代差结果展示p1与p2的不同的辈分关系。

（7）某成员添加孩子

输入父亲名称，若查找失败则报错提醒。依次输入所添加孩子的基本信息，为其父节点指针赋值，同时在父亲的孩子列表中添加该孩子。

（8）删除某成员（若其还有后代，则一并删除）。

输入名称，若查找失败则报错提醒。采用后序遍历方式，先删除指定成员的所有孩子节点后再删除自身。

（9）修改某成员信息。

输入名称，若查找失败则报错提醒。依次输入修改信息后保存。

2. 各模块的算法描述

（1）文件读写

void GMS::readFile()

{

//打开文件输入流

ifstream fin(GMS::filePath);

if (fin.is\_open() == false)

{

cout << "文件打开失败" << endl;

return;

}

int id = -1;

string name = "";

string birthday = "";

bool married = false;

string address = "";

bool live = false;

string deathDate = "";

string fatherName = "";

string line;

stringstream ss;

if (fin.eof() == false)

{

//首行为表头

getline(fin, line);

}

while (fin.eof() == false)

{

getline(fin, line);

//若为空行则结束读取

if (line.length() == 0)

{

break;

}

//将非空行读入字符串流并为各变量依次赋值

ss << line;

ss >> id >> name >> fatherName >> birthday >> married >> live >> deathDate >> address;

//考虑到姓名、地址可能会有空格的情况，故将姓名、地址用[]包住

name = name.substr(1, name.length() - 2);

fatherName = fatherName.substr(1, fatherName.length() - 2);

address = address.substr(1, address.length() - 2);

//考虑到死亡日期在是否健在值为true时为空字符串，使用-进行占位

deathDate = deathDate == "-" ? "" : deathDate;

Person\* p = new Person();

p->setId(id);

p->setName(name);

p->setBirthday(birthday);

p->setMarried(married);

p->setAddress(address);

p->setLive(live);

p->setDeathDate(deathDate);

//先序遍历的第一个节点即为祖先节点

if (this->ancestor == NULL)

{

this->ancestor = p;

}

Person\* father = this->searchByName(this->ancestor, fatherName);

p->setFather(father);

if (father != NULL)

{

father->addChild(p);

}

//更新当前id生成器的最新值

IdGenerator::update(id);

}

//关闭文件输入流

fin.close();

}

void GMS::writeFile()

{

//打开文件输出流

ofstream fout(GMS::filePath);

if (fout.is\_open() == false)

{

cout << "文件打开失败" << endl;

return;

}

//输出表头行

fout << "ID\t" << "姓名\t\t" << "父亲\t\t" << "出生日期\t" << "婚否\t"

<< "健在否\t" << "死亡日期\t" << "地址\t" << endl;

//采用先序遍历方式将所有成员输出至指定文件中

this->writeFile(this->ancestor, fout);

//关闭文件输出流

fout.close();

}

//采用先序遍历方式将所有成员输出至指定文件中

void GMS::writeFile(Person\* root, ofstream& fout)

{

if (root == NULL)

{

return;

}

//考虑到姓名、地址可能会有空格的情况，故将姓名、地址用[]包住

//考虑到死亡日期在是否健在值为true时为空字符串，使用-进行占位

fout << root->getId() << "\t[" << root->getName() << "]\t\t["

<< (root->getFather() == NULL ? "" : root->getFather()->getName()) << "]\t\t"

<< root->getBirthday() << "\t" << root->isMarried() << "\t" << root->isLive() << "\t"

<< (root->getDeathDate() == "" ? "-" : root->getDeathDate()) << "\t["

<< root->getAddress() << "]\t" << endl;

vector<Person\*>& children = root->getChildren();

for (vector<Person\*>::const\_iterator iter = children.cbegin(); iter != children.cend(); iter++)

{

this->writeFile(\*iter, fout);

}

}

（2）家谱展示

void GMS::displayGenealogy()

{

//显示层级结构

this->showLevel();

//显示表头

this->showTableHeader();

//显示所有成员信息列表

this->showAllPersons(this->ancestor);

}

void GMS::showLevel()

{

if (this->ancestor == NULL)

{

return;

}

//通过队列实现层序遍历

queue<Person\*> queue;

queue.push(this->ancestor);

while (queue.empty() == false)

{

int num = queue.size();

for (int i = 0; i < num; i++) {

Person\* p = queue.front();

queue.pop();

cout << p->getName() << (p->getFather() == NULL ? "" : "（父：" + p->getFather()->getName() + "）") << "\t";

vector<Person\*>& children = p->getChildren();

for (vector<Person\*>::const\_iterator iter = children.cbegin(); iter != children.cend(); iter++)

{

queue.push(\*iter);

}

}

cout << endl;

}

cout << endl;

}

void GMS::showAllPersons(Person\* root)

{

if (root == NULL)

{

return;

}

root->display();

vector<Person\*>& children = root->getChildren();

for (vector<Person\*>::const\_iterator iter = children.cbegin(); iter != children.cend(); iter++)

{

this->showAllPersons(\*iter);

}

}

（3）按代数查询

void GMS::displayGeneration()

{

int gen = -1;

cout << "请输入第几代：";

cin >> gen;

while (gen <= 0)

{

cout << "无效数字，请重新输入：";

cin >> gen;

}

vector<Person\*> persons;

this->searchByGeneration(this->ancestor, gen, 1, persons);

cout << "查询结果：" << endl;

if (persons.size() == 0)

{

cout << "无" << endl;

}

else

{

this->showTableHeader();

for (vector<Person\*>::const\_iterator iter = persons.cbegin(); iter != persons.cend(); iter++)

{

(\*iter)->display();

}

}

}

void GMS::searchByGeneration(Person\* root, int tarGen, int curGen, vector<Person\*>& persons)

{

//采用先序遍历方式当层数为目标值时添加至容器中

if (root == NULL)

{

return;

}

if (tarGen == curGen)

{

persons.push\_back(root);

return;

}

vector<Person\*>& children = root->getChildren();

for (vector<Person\*>::const\_iterator iter = children.cbegin(); iter != children.cend(); iter++)

{

this->searchByGeneration(\*iter, tarGen, curGen + 1, persons);

}

}

（4）按姓名查询

void GMS::searchByName()

{

string name = "";

cout << "姓名：";

// getchar();

cin >> name;// getline(cin, name);

Person\* p = this->searchByName(this->ancestor, name);

if (p == NULL)

{

cout << "无此成员！" << endl;

return;

}

cout << "本人信息：" << endl;

this->showTableHeader();

p->display();

cout << "父亲信息：" << endl;

if (p->getFather() != NULL)

{

this->showTableHeader();

p->getFather()->display();

}

else

{

cout << "无" << endl;

}

cout << "孩子信息：" << endl;

vector<Person\*>& children = p->getChildren();

if (children.size() == 0)

{

cout << "无" << endl;

}

else

{

this->showTableHeader();

for (vector<Person\*>::const\_iterator iter = children.cbegin(); iter != children.cend(); iter++)

{

(\*iter)->display();

}

}

}

Person\* GMS::searchByName(Person\* root, string name)

{

//采用先序遍历方式查找指定姓名的成员并返回，若查找失败返回NULL

Person\* res = NULL;

if (root == NULL)

{

return NULL;

}

if (name == root->getName())

{

return root;

}

vector<Person\*>& children = root->getChildren();

for (vector<Person\*>::const\_iterator iter = children.cbegin(); iter != children.cend(); iter++)

{

res = this->searchByName(\*iter, name);

if (res != NULL)

{

break;

}

}

return res;

}

（5）按出生日期查询

void GMS::searchByBirthday()

{

string birthday = "";

cout << "出生日期：";

cin >> birthday;

vector<Person\*> persons;

searchByBirthday(this->ancestor, birthday, persons);

cout << "查询结果：" << endl;

if (persons.size() == 0)

{

cout << "无" << endl;

}

else

{

this->showTableHeader();

for (vector<Person\*>::const\_iterator iter = persons.cbegin(); iter != persons.cend(); iter++)

{

(\*iter)->display();

}

}

}

void GMS::searchByBirthday(Person\* root, string birthday, vector<Person\*>& persons)

{

//采用先序遍历方式当出生日期与参数相等时添加至容器中

if (root == NULL)

{

return;

}

if (birthday == root->getBirthday())

{

persons.push\_back(root);

}

vector<Person\*>& children = root->getChildren();

for (vector<Person\*>::const\_iterator iter = children.cbegin(); iter != children.cend(); iter++)

{

this->searchByBirthday(\*iter, birthday, persons);

}

}

（6）输入两人姓名，确定其关系

void GMS::getRelationshipByNames()

{

string name1 = "";

cout << "姓名1：";

// getchar();

cin >> name1;// getline(cin, name);

Person\* p1 = this->searchByName(this->ancestor, name1);

if (p1 == NULL)

{

cout << "无此成员！" << endl;

return;

}

string name2 = "";

cout << "姓名2：";

// getchar();

cin >> name2;// getline(cin, name);

Person\* p2 = this->searchByName(this->ancestor, name2);

if (p2 == NULL)

{

cout << "无此成员！" << endl;

return;

}

//计算p1代数

int dis1 = 0;

Person\* pp1 = p1;

while (pp1->getFather() != NULL)

{

dis1++;

pp1 = pp1->getFather();

}

//计算p2代数

int dis2 = 0;

Person\* pp2 = p2;

while (pp2->getFather() != NULL)

{

dis2++;

pp2 = pp2->getFather();

}

//计算代差

int diff = dis2 - dis1;

if (diff < -1)

{

printf("%s比%s小%d辈\n", name1.c\_str(), name2.c\_str(), -1 \* diff);

}

else if (diff == -1)

{

if (p1->getFather() == p2)

{

printf("%s是%s的孩子\n", name1.c\_str(), name2.c\_str());

}

else

{

printf("%s比%s小%d辈\n", name1.c\_str(), name2.c\_str(), -1 \* diff);

}

}

else if (diff == 0)

{

if (p2 == p1)

{

printf("%s与%s是同一个人\n", name1.c\_str(), name2.c\_str());

}

else

{

printf("%s与%s同辈\n", name1.c\_str(), name2.c\_str());

}

}

else if (diff == 1)

{

if (p2->getFather() == p1)

{

printf("%s是%s的父亲\n", name1.c\_str(), name2.c\_str());

}

else

{

printf("%s比%s大%d辈\n", name1.c\_str(), name2.c\_str(), diff);

}

}

else

{

printf("%s比%s大%d辈\n", name1.c\_str(), name2.c\_str(), diff);

}

}

（7）某成员添加孩子

void GMS::addChildren()

{

string name = "";

cout << "父亲姓名：";

// getchar();

cin >> name;// getline(cin, name);

Person\* p = this->searchByName(this->ancestor, name);

if (p == NULL)

{

cout << "无此成员！" << endl;

return;

}

cout << "请输入孩子信息" << endl;

Person\* child = new Person();

this->modifyPerson(child);

p->addChild(child);

cout << "添加成功" << endl;

}

（8）删除某成员（若其还有后代，则一并删除）。

void GMS::remove()

{

string name = "";

cout << "姓名：";

//getchar();

cin >> name;// getline(cin, name);

Person\* p = this->searchByName(this->ancestor, name);

if (p == NULL)

{

cout << "无此成员！" << endl;

return;

}

if (p->getFather() != NULL)

{

//将指定成员从父亲的孩子列表中删除

vector<Person\*>& children = p->getFather()->getChildren();

for (vector<Person\*>::const\_iterator iter = children.cbegin(); iter != children.cend(); )

{

if ((\*iter) == p)

{

iter = children.erase(iter);

}

else

{

iter++;

}

}

}

//删除自身与孩子列表

this->remove(p);

cout << "删除成功！" << endl;

}

void GMS::remove(Person\* root)

{

if (root == NULL)

{

return;

}

vector<Person\*>& children = root->getChildren();

for (vector<Person\*>::const\_iterator iter = children.cbegin(); iter != children.cend(); )

{

Person\* p = \*iter;

//从孩子列表中删除当前孩子

iter = children.erase(iter);

//删除孩子自身与孩子的孩子节点列表

this->remove(p);

}

//确保孩子节点全部删除

children.clear();

//删除自身

delete root;

}

（9）修改某成员信息。

void GMS::update()

{

string name;

cout << "姓名：";

// getchar();

cin >> name;// getline(cin, name);

Person\* p = this->searchByName(this->ancestor, name);

if (p == NULL)

{

cout << "无此成员！" << endl;

return;

}

cout << "请输入修改后信息" << endl;

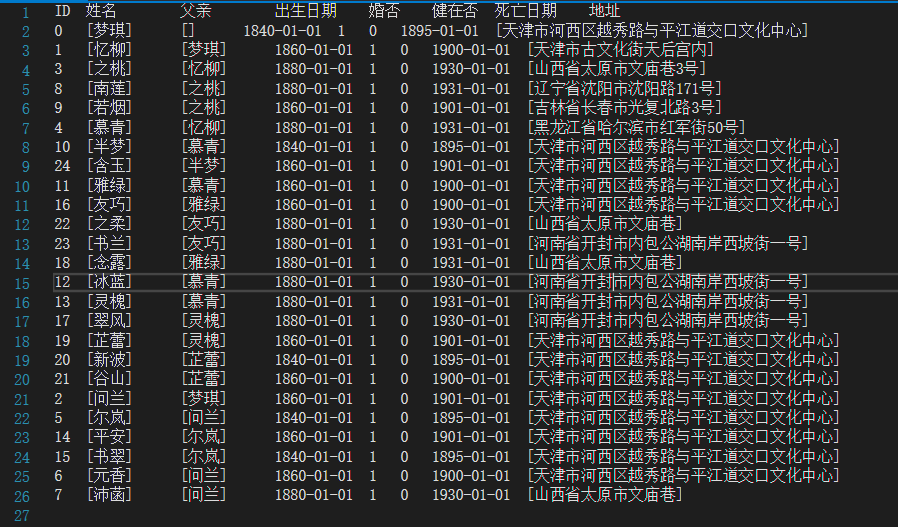
this->modifyPerson(p);

cout << "修改成功！" << endl;

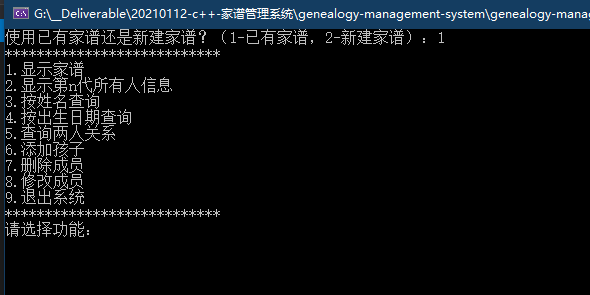
}

3. 运行测试结果

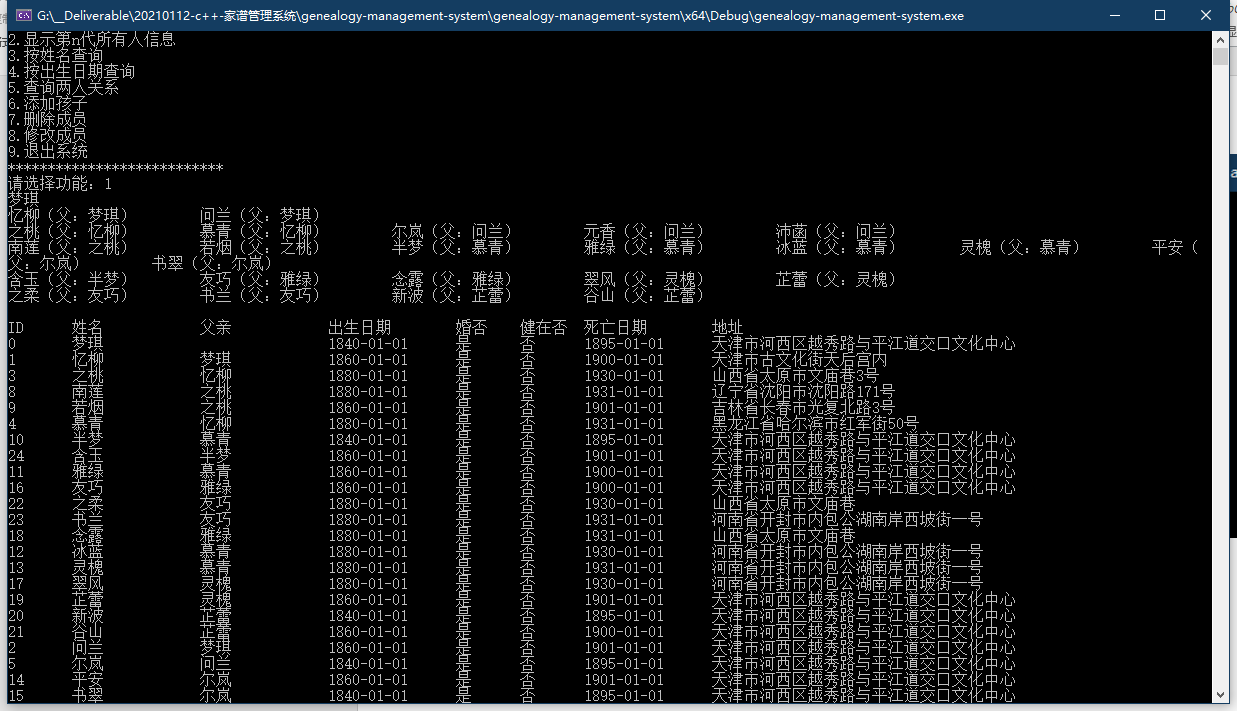
（1）输入文件



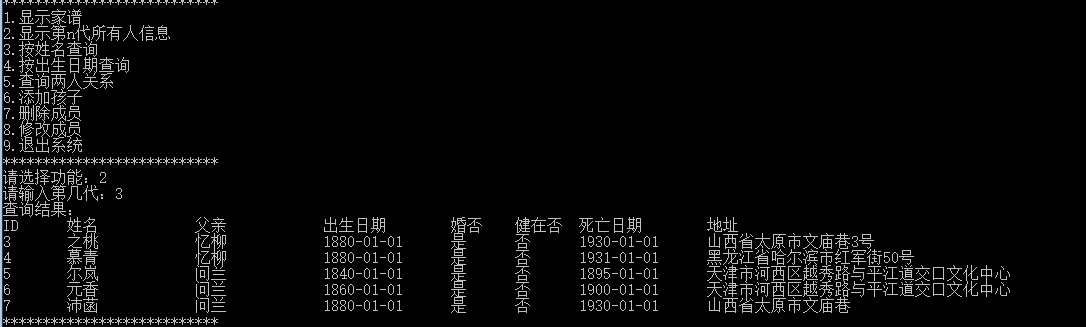
（2）程序启动



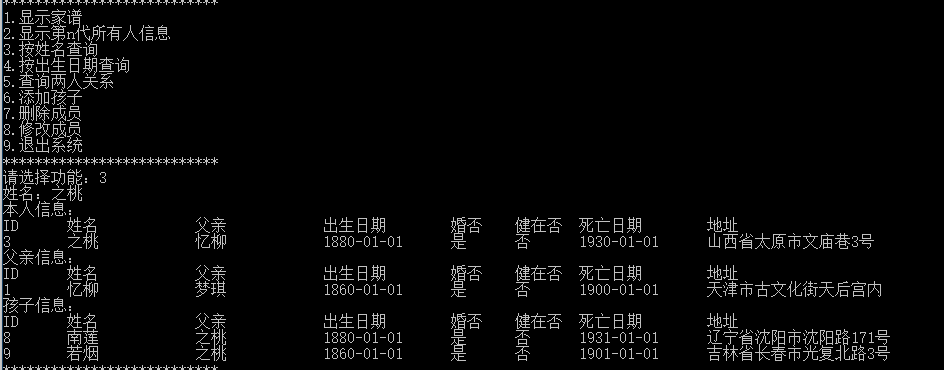
（3）以图形方式显示家谱。



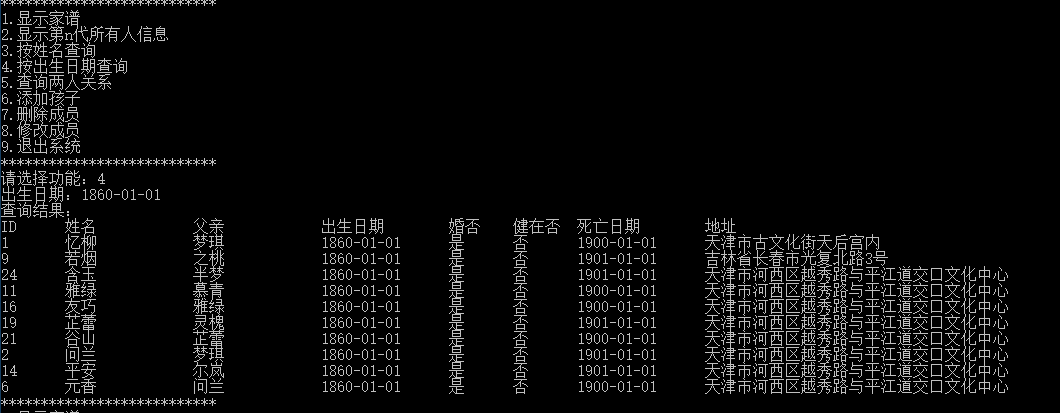
（4）显示第n 代所有人的信息。



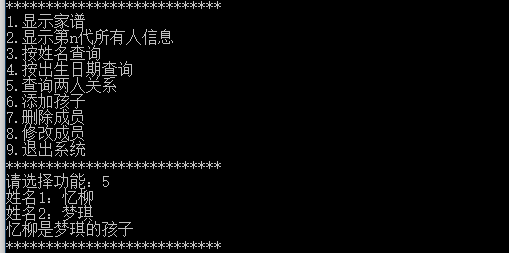
（5）按照姓名查询，输出成员信息（包括其本人、父亲、孩子的信息）。



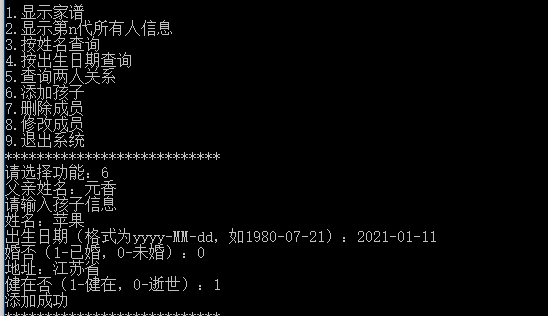
（6）按照出生日期查询成员名单。



（7）输入两人姓名，确定其关系。



（8）某成员添加孩子。



（9）删除某成员（若其还有后代，则一并删除）。



（10）修改某成员信息。



# 六、调试分析

这次的综合性实验使我了解到，平时对知识的积累相当重要，同时也要注重课上老师的讲解，老师在课上的延伸是课本上所没有的，这些知识对于我们对程序的编写有很大的作用，同时，编程也要求我们有足够的耐心，细细推敲。越着急可能就越无法得到我们想要的结果，遇到不会的问题要多多请教，知识是在实践与向别人请教的过程中积累的，所以问是至关重要的，只要肯下功夫很多东西都是可以完成的。

感触最深的点就在于c++语言的深浅拷贝以及引用这种特殊的数据类型。引用是C++引入的新语言特性，是C++常用的一个重要内容之一，正确、灵活地使用引用，可以使程序简洁、高效。引用就是某一变量（目标）的一个别名，对引用的操作与对变量直接操作完全一样。在删除成员的方法实现过程中，一开始我是使用的普通的临时变量来接收孩子列表，删除操作结束后却发现原有的孩子列表的长度并没有发生任何变化，经过多次排查，终于确定是局部变量的拷贝问题所导致的，改为使用引用后问题就得到了解决。

# 七、参考文献

1 严蔚敏．《数据结构》（C语言版），清华大学出版社，2007

2 Donald E.Knuth. 《The Art of Computer Programming》，Addison-Wesley Professional，1997

3 傅清祥，王晓东. 《算法与数据结构，电子工业出版社，2001