CSC 122 001 Computer Science II

Julius Ranoa

Quiz 9 Program 19-5 Documentation.

Instructions: (1) Document the Program, (2) Refactor the source to the OOP Paradigm, (3) Change the family tree, and (4) Include a screenshot of the command window.

This documentation will be as follows:

1. Overview and General Algorithm
2. Class Definition (Source Code)
3. Methods and Intended Behaviors (including Source Code)
4. Sample Driver Program (with custom Family Tree)
5. Appendix: Actual Source Code.

**OVERVIEW AND GENERAL ALGORIGHTM**

The source defines a *Person* class that stores the following information: (1) the name of the person, (2) the person’s gender (in a enumeration data type), (3) a vector of pointers of the person’s parents, and (4) a vector of pointers of the person’s children.

Helper functions will be added to facilitate the addition of parents and children. An overloaded output stream operator is also given for printing a summary of the *Person*’s information.

**CLASS DEFINITION (Person.h)**

#include **<vector>**#include **<string>**#include **<iostream>**#include **<cstdlib>  
using namespace** std;  
  
**enum** Gender{***male***, ***female***};  
  
*// Person class represents a person participating in a genealogy.***class** Person  
{  
 string name;  
 Gender gender;  
 vector<Person \*> parents;  
 vector<Person \*> children;  
 **void** addParent(Person \*p){ parents.push\_back(p); }  
**public**:  
 Person (string name, Gender g)  
 {  
 **this**->name = name;  
 gender = g;  
 }  
 Person \*addChild(string name, Gender g);  
 Person \*addChild(Person \*p);  
  
 **friend** ostream &**operator** << (ostream &out, Person p);  
  
 *// Member functions for getting various Person info* string getName() **const**{ **return** name; };  
 Gender getGender() **const**{ **return** gender; };  
 **int** getNumChildren() **const**{ **return** children.size(); }  
 **int** getNumParents() **const**{ **return** parents.size(); }  
 Person \*getChild(**int** k) **const** ;  
 Person \*getParent(**int** k) **const**;  
};

**METHODS AND INTENDED BEHAVIORS (including source code from Person.cpp)**

*\* Standard accessors are not included in the list here.*

1. ***Person* Constructor**

Method Declaration/Definition:



This is a standard constructor object.

1. **Adding a parent**

Method Declaration/Definition:



This accepts a pointer of the *Person* object and adds it the vector of parents.

1. **Adding a child without existing Person object**

Method Signature: Person \*addChild(string name, Gender g);

This method creates a new *Person* object for the child and passes the original arguments for name and gender. The *Person* object calling this method (i.e. the parent) is added as a parent to the new object. A pointer of the new object is also added to the parent object’s children vector. A pointer to the new child object is returned.

Method Definition:



1. **Adding a child with exiting *Person* object.**

Method Signature: Person \*addChild(Person \*p);

This method assumes that a child object has already been created. This is intended to be used once the earlier method for adding children has been called (i.e. adding the same child to another parent).

The method does two things: (1) adding the object that calls the method (i.e. the parent object) as the parent of the child object (i.e. the argument), and (2) adding the argument (i.e. the child pointer) to the children vector of the parent object.

Method Definition:



1. **The overloaded output stream operator (<<)**

Method Signature: **friend** ostream &**operator** << (ostream &out, Person p);

This method prints out the information of the *Person* object in a way reminiscent of a mark-up language. It does the following:

1. Prints out the starting tags for the Person with the name. (i.e. <Person … >)
2. If the parent vector is not empty, add the starting tag for parent (i.e. <parents>), print the names of all the parents, and end the tag (i.e. </parents>)
3. If the children vector is not empty, add the starting tag for children (i.e. <children>), print the names of all the children, and end the tag (i.e. </children>)
4. Print the end tag for the Person (i.e. </Person>)

Method Definition:



1. **Accessor method for children with index *k***

Method Signature: Person \*getChild(**int** k) **const** ;

Since the object allows for the addition of multiple children, there must be a way to access the children objects one by one. This method does that by identifying each child with an index *k*, which would be the child object’s position in the vector. If the index passed is out of bounds (i.e. negative || greater than or equal to the number of children), the program exits with an exit code of 1.

Method Definition:



1. **Accessor method for parent with index *k***

Method Signature:

The implementation for this is similar for Person \*getChild(**int** k) **const** ;. See previous item.

Method Definition:



**SAMPLE DRIVER PROGRAM (with custom family tree)**

The following family tree is used:

(Molly and Arthur) -> Ron and Ginny

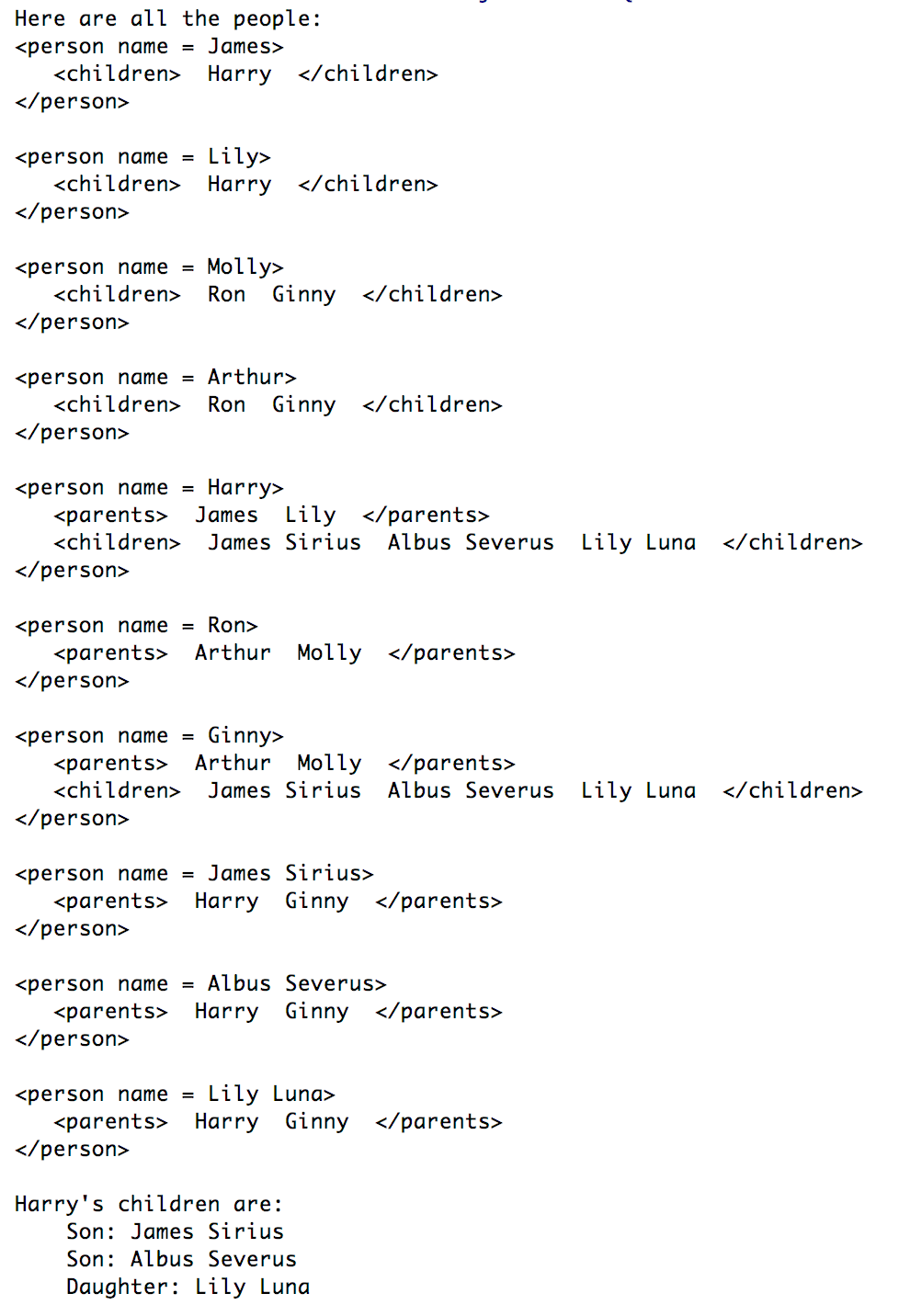
(James and Lily) -> Harry

(Ginny and Harry) -> James Sirius, Albus Severus, and Lily Luna.

The family tree implementation is as follows (main.cpp)

*// 1st Generation* Person james(**"James"**, ***male***);  
 Person lily(**"Lily"**, ***female***);  
  
 Person arthur(**"Arthur"**, ***male***);  
 Person molly(**"Molly"**, ***female***);  
  
 *// 2nd Generation* Person \* harry = james.addChild(**"Harry"**, ***male***);  
 lily.addChild(harry);  
 Person \* ron = arthur.addChild(**"Ron"**, ***male***);  
 molly.addChild(ron);  
 Person \* ginny = arthur.addChild(**"Ginny"**, ***female***);  
 molly.addChild(ginny);  
  
 *// The Grandchildren* Person \* james2nd = harry->addChild(**"James Sirius"**, ***male***);  
 Person \* albus = harry->addChild(**"Albus Severus"**, ***male***);  
 Person \* lily2nd = harry->addChild(**"Lily Luna"**, ***female***);  
 ginny->addChild(james2nd);  
 ginny->addChild(albus);  
 ginny->addChild(lily2nd);  
  
 *// Output all people.* cout << **"Here are all the people: \n"**;  
 cout << james << **"\n"** << lily << **"\n"**;  
 cout << molly << **"\n"** << arthur << **"\n"**;  
 cout << \*harry << **"\n"** << \*ron << **"\n"** << \*ginny << **"\n"**;  
 cout << \*james2nd << **"\n"** << \*albus << **"\n"** << \*lily2nd << **"\n"**;  
  
 *// Print out the children of Harry* cout << **"Harry's children are: \n"**;  
 **for** (**int** i = 0; i < harry->getNumChildren(); i++) {  
 Person \* temp = harry->getChild(i);  
 **switch** (temp->getGender()) {  
 **case *male***:  
 cout << **" Son: "** << temp->getName() << **"\n"**;  
 **break**;  
 **case *female***:  
 cout << **" Daughter: "** << temp->getName() << **"\n"**;  
 **break**;  
 }  
 }

Screenshot of runtime:



**APPENDIX: ACTUAL SOURCE CODE:**

Three files are included: (1) main.cpp, (2) Person.h, and (3) Person.cpp

**main.cpp**

*// This program uses a generalization of binary trees to build  
// genealogy trees.*#include **"Person.h"  
using namespace** std;  
  
**int** main(**int** argc, **char** \*\* argv) {  
 *// 1st Generation* Person james(**"James"**, ***male***);  
 Person lily(**"Lily"**, ***female***);  
  
 Person arthur(**"Arthur"**, ***male***);  
 Person molly(**"Molly"**, ***female***);  
  
 *// 2nd Generation* Person \* harry = james.addChild(**"Harry"**, ***male***);  
 lily.addChild(harry);  
 Person \* ron = arthur.addChild(**"Ron"**, ***male***);  
 molly.addChild(ron);  
 Person \* ginny = arthur.addChild(**"Ginny"**, ***female***);  
 molly.addChild(ginny);  
  
 *// The Grandchildren* Person \* james2nd = harry->addChild(**"James Sirius"**, ***male***);  
 Person \* albus = harry->addChild(**"Albus Severus"**, ***male***);  
 Person \* lily2nd = harry->addChild(**"Lily Luna"**, ***female***);  
 ginny->addChild(james2nd);  
 ginny->addChild(albus);  
 ginny->addChild(lily2nd);  
  
 *// Output all people.* cout << **"Here are all the people: \n"**;  
 cout << james << **"\n"** << lily << **"\n"**;  
 cout << molly << **"\n"** << arthur << **"\n"**;  
 cout << \*harry << **"\n"** << \*ron << **"\n"** << \*ginny << **"\n"**;  
 cout << \*james2nd << **"\n"** << \*albus << **"\n"** << \*lily2nd << **"\n"**;  
  
 *// Print out the children of Harry* cout << **"Harry's children are: \n"**;  
 **for** (**int** i = 0; i < harry->getNumChildren(); i++) {  
 Person \* temp = harry->getChild(i);  
 **switch** (temp->getGender()) {  
 **case *male***:  
 cout << **" Son: "** << temp->getName() << **"\n"**;  
 **break**;  
 **case *female***:  
 cout << **" Daughter: "** << temp->getName() << **"\n"**;  
 **break**;  
 }  
 }  
  
 **return** 0;  
}

**Person.h**

#ifndef **QUIZ9\_PR19\_5\_DOCUMENTATION\_SOURCEFILE\_PERSON\_H**#define **QUIZ9\_PR19\_5\_DOCUMENTATION\_SOURCEFILE\_PERSON\_H**#include **<vector>**#include **<string>**#include **<iostream>**#include **<cstdlib>  
using namespace** std;  
  
**enum** Gender{***male***, ***female***};  
  
*// Person class represents a person participating in a genealogy.***class** Person  
{  
 string name;  
 Gender gender;  
 vector<Person \*> parents;  
 vector<Person \*> children;  
 **void** addParent(Person \*p){ parents.push\_back(p); }  
**public**:  
 Person (string name, Gender g)  
 {  
 **this**->name = name;  
 gender = g;  
 }  
 Person \*addChild(string name, Gender g);  
 Person \*addChild(Person \*p);  
  
 **friend** ostream &**operator** << (ostream &out, Person p);  
  
 *// Member functions for getting various Person info* string getName() **const**{ **return** name; };  
 Gender getGender() **const**{ **return** gender; };  
 **int** getNumChildren() **const**{ **return** children.size(); }  
 **int** getNumParents() **const**{ **return** parents.size(); }  
 Person \*getChild(**int** k) **const** ;  
 Person \*getParent(**int** k) **const**;  
};  
  
#endif *//QUIZ9\_PR19\_5\_DOCUMENTATION\_SOURCEFILE\_PERSON\_H*

**Person.cpp**

#include **"Person.h"***//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  
// Create a child with specified name and gender, and \*  
// set one of the parents to be this person. \*  
// Add the new child to the list of childfen for this person \*  
//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\**Person \*Person::addChild(string name, Gender g)  
{  
 Person \*child = **new** Person(name, g);  
 child->addParent(**this**); *// I am a parent of this child* children.push\_back(child); *// This is one of my children* **return** child;  
}  
  
*//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  
// Add a child to the list of children for this person \*  
//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\**Person \*Person::addChild(Person\* child)  
{  
 child->addParent(**this**); *// I am a parent of this child* children.push\_back(child); *// This is one of my children* **return** child;  
}  
  
*//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  
// Return a pointer to the specified parent \*  
//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\**Person \*Person::getParent(**int** k) **const**{  
 **if** (k < 0 || k >= parents.size())  
 {  
 cout << **"Error indexing parents vector."** << endl;  
 exit(1);  
 }  
 **return** parents[k];  
}  
  
*//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  
// Return a pointer to a specified child \*  
//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\**Person \*Person::getChild(**int** k) **const**{  
 **if** (k < 0 || k >= children.size())  
 {  
 cout << **"Error indexing children's vector."** << endl;  
 exit(1);  
 }  
 **return** children[k];  
}  
  
*//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  
// Overloaded stream output operator \*  
//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\**ostream & **operator**<<(ostream & out, Person p)  
{  
 out << **"<person name = "** << p.name << **">"** << **'\n'**;  
 **if** (p.parents.size() > 0)  
 out << **" <parents>"** << **' '**;  
 **for** (**int** k = 0; k < p.parents.size(); k++)  
 {  
 out << **" "** << p.parents[k]->name << **' '**;  
 }  
 **if** (p.parents.size() > 0)  
 out << **" </parents>"** << **"\n"**;  
 **if** (p.children.size() > 0)  
 out << **" <children>"** << **' '**;  
 **for** (**int** k = 0; k < p.children.size(); k++)  
 {  
 out << **" "** << p.children[k]->name << **' '**;  
 }  
 **if** (p.children.size() > 0)  
 out << **" </children>"** << **"\n"**;  
 out << **"</person>"** << **"\n"**;  
 **return** out;  
}