ECOTE – Final Documentation

Date: 29.05.2021

Semester: 21L

Author and Group: Aleksandra Wyrostkiewicz, group 103

Subject: 13. Program reading C++ code and constructing class inheritance tree.

I. General overview and assumptions

The task is to design a program reading C++ code, and constructing inheritance tree of its classes. The program should take the file name as an input and return the inheritance tree as an output in the console. If there are many trees, all of them should be written out, also the ones that has no children.

Code reading assumptions

It is assumed that the C++ code is in **one file** and there are **no errors in it**, as well as **basic clean coding rules** are applied. Program should scan the file looking for following structures:

Class declaration:

class class-name base-clause (optional)

Where:

class-name	-	**the name of the class that is being defined.**
base-clause	-	optional clause for name of parent class and the model of inheritance used (see below)

Base-clause:

: access-specifier (optional) virtual-specifier (optional) parent-class-specifier

Where:

access-specifier	-	**one of private, public, or protected**
virtual-specifier	-	the keyword virtual
parent-class-specifier	-	name of parent class

access-specifier and *virtual-specifier* may appear in any order

Adding appropriate class as a root or leaf node, depending on structure found. Program should not create new trees for classes already encountered, as in forward declaration. Additionally, there exists an assumption that **while single class can have many children, they only one parent is allowed**.

File reading assumptions

The program should be run with no command line argument (for base tests) or with any number of them, every of them being either name of file containing C++ code, in case the file is in the same folder as program, or full directory path to the file. The file should have extension ".cpp".

II. Functional requirements

The general purpose of inheritance tree constructor is to create and draw all inheritance class trees emerging from the given file. This means that a class library is needed, in which all the defined classes, encountered in the source text, are stored and ordered. This data structure is further described in the *Data structures* section.

The tree constructor scans every line of given file, saving classes in class library. The key words for the program are:

KEY WORD		MEANING
class	-	Start of class definition
:	-	Start of base-clause
virtual	-	Virtual specifier ignored by the program
private / public / protected	-	Access specifier ignored by the program
{	-	End of class definition
;	-	End of forward declaration

If the class has no base-clause, it is saved as a root to which other classes can be added as a leaf. In the opposite case, program adds a class as a leaf to nodes of all classes listed in base-clause. If a class is encountered again, meaning it already exists in class library (e.g., because of forward declaration), it is either added as a leaf to an existing class tree or ignored, depending on existence of base-clause.

The constructor needs developed error/warning system, as the only input given by the user is file path. The precise description of it is in the *Input/output description* section.

III. Implementation

General architecture

The program consists of two main modules:

TreeConstructor Module – main, responsible for the Command Line Interface, file reading and constructing tree list

Tree Class – container-class used by other modules

The class diagram and description are presented below:

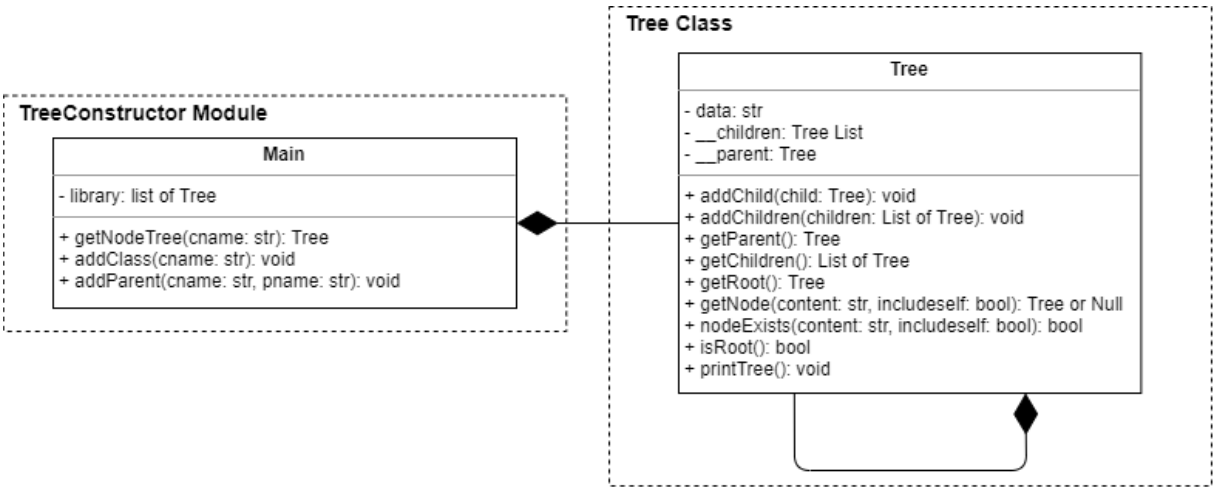

Figure 1. Class Diagram

Main:

library: list of Tree	**List of Tree objects**
getNodeTree(cname: str): Tree or Null	Function returns Tree object with specified data, if there is none, returns null
addClass(cname: str): void	If Tree library does not include object with specified name, function creates new Tree object in the library
addParent(cname: str, pname: str): void	If Tree library includes parent object (pname), function adds existing node (cname) as its child, or creates new child node if it does not exist yet

Class Tree:

data: str	Node name / data
__children: Tree List	List of Tree objects, children of the node. Default empty
__parent: Tree or Null	Parent Tree object. Default null
addChild(child: Tree): void	Function adding child Tree object to self
addChildren(children: List of Tree): void	Function adding multiple child Tree objects to self
getParent(): Tree	Function returning Tree parent of object
getChildren(): List of Tree	Function returning Tree vector of children
getRoot(): Tree	Function returning root of Tree (top parent)
getNode(content: str, includeself: bool): Tree or Null	Function returning Tree object with specified content or null if not found. Default includeself equal to True
NodeExists(content: str, includeself: bool): bool	Function returning bool of existence of Tree with equal content. Default includeself equal to True
isRoot(): bool	Returning True for root node and false for anything else
printTree(): void	Function responsible for pretty drawing of the tree

Data structures

STRUCTURE		DESCRIPTION
TreeLibrary	-	Vector containing Tree objects
Tree	-	Tree object composed of Tree objects. The tree hierarchical tree structure, with a root value and subtrees of children with a parent node, represented as a set of linked nodes.

Module descriptions

The most important part of the working module, Tree Constructor - which is performing the most crucial operations, is presented below on the Figure 2. There is no flowchart for Tree Class, as it is just a set of functions constructing class.

Tree Constructor Module:

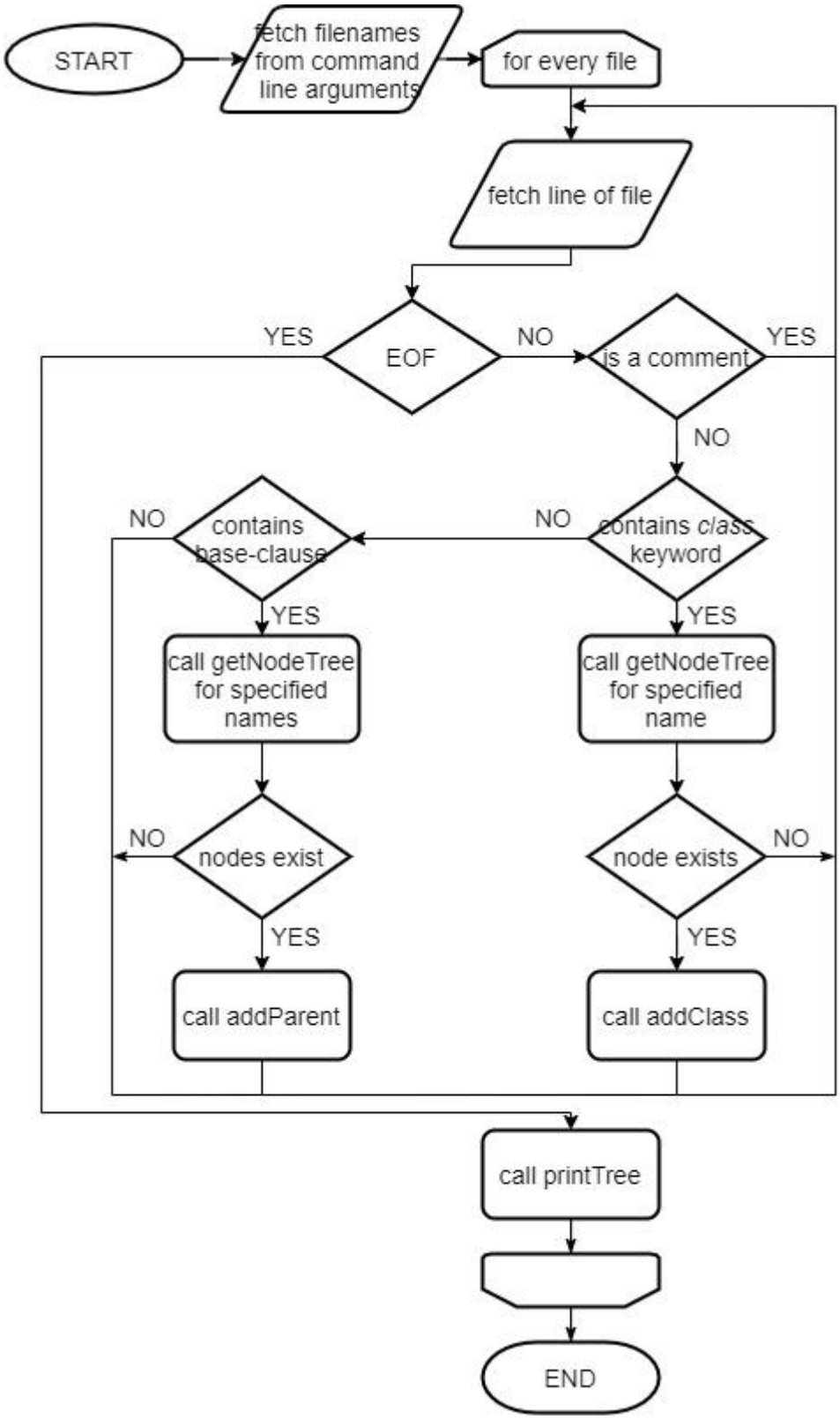

Figure 2. Tree Constructor Module Flowchart

Input/output description

The program uses command line parameters as an input, which is path to the C++ code file. Possible command line parameters are in form “filename.cpp” or “path/to/directory.filename.cpp”. Multiple command line arguments are possible. Running without command line parameters executes two files “example1.cpp” and “example2.cpp” as a test case.

As the only possible error, assuming that the code is correct, is nonexistent file for given path, the error handling can be summarized into checking if the file exists in the given location and sending a “-File not accessible” message for every inaccessible file.

Others

The project is implemented in Python, and the documentation can be found on my GitHub: https://github.com/aleks-arya/CPP-Inheritance-Tree-Constructor

IV. Functional test cases

The following test cases were carried out:

Input: `py tree_constructor.py`
Output:

```
File: example1.cpp
A
 |___ AA
 |___ B
       |___ BB
             |___ BBB
             |___ Badsf
D
 |___ DD
 |    |___ DDD
 |___ DD2

File: example2.cpp
A
 |___ AA
 |    |___ D
 |          |___ DD
 |          |    |___ DDD
 |          |___ DD2
 |___ B
       |___ BB
             |___ BBB
             |___ Badsf
```

Input: `py tree_constructor.py "F:\Docs\asd.cpp" "1231231.cpp"`
Output:

```
File: F:\Docs\asd.cpp
-File not accessible

File: 1231231.cpp
-File not accessible
```

Input: `py tree_constructor.py "example1.cpp" "asdf.cpp"`
Output:

```
File: example1.cpp
A
 |___ AA
 |___ B
       |___ BB
             |___ BBB
             |___ Badsf
D
 |___ DD
 |    |___ DDD
 |___ DD2

File: asdf.cpp
-File not accessible
```

Input: `py tree_constructor.py "random_code.cpp"`
Output:

```
File: random_code.cpp
root
 |___ child
 |    |___ childchild
 |          |___ childchildchild
 |          |___ childchildsequel
 |___ seconndchild

anotherRoot

yetanotherRoot
 |___ childofsth
```