CTutor

Tiago Aguiar
tiago.afonso.aguiar@ist.utl.pt

Instituto Superior Técnico, Lisboa, Portugal

November 2014

Abstract

CTutor is a program visualization tool for the programming language, C. As the name implies, CTutor serves as tutor for learning C. The common objective of program visualization is to enhance the comprehension of the several elements of a certain programming language, such as types of variables, function's arguments and outputs.

Program visualization tools are an useful complement to the learning process of a new programming language, replacing the typical images drew on blackboards, or the figures made in PowerPoint slides. In case of C, these images are often used to explain sorting algorithms, pointers and structures. With CTutor, the visualizations are produced from the source code, written by the student or the professor. This source code is executed by CTutor's backend (execution environment) and the visualizations are shown in the frontend (visualizer).

CTutor also presents a change in the the typical architecture of program visualization tools. Program visualization tools are composed by an execution environment and a visualizer, and usually are strongly coupled, thus hindering the creation of new tools that reuse one of these components in another tool. With this major change in the architecture, it is possible to use CTutor's execution environment and create a new visualizer and adapting it to student's needs. Or, it is possible to change the execution environment to execute another programming language and use it with the CTutor's visualizer.

Keywords: Learning, Visualization, Students, C language, Architecture

1. Introduction

This document introduces CTutor, a program visualization tool for the C programming language [1].

Nowadays, each person can learn programming in three different ways: self learning, e-learning or a programming course in a school. As a student, the learning process of a new programming language follows an approach bottom-up, starting with simple examples ("Hello World") with the primitive data types and then the complexity increases gradually.

In terms of tools used in the learning process, the student starts with a simple editor to write the source code and when he is more comfortable with the new learned language, is common to use an integrated development environment (IDE), such as Eclipse [2] or NetBeans [3].

The described process is also complemented with some graphical illustrations for certain examples. These images are used with the objective of clarifying some doubts about the use of some complex types (e.g. Lists, Arrays, Dictionaries) or some paradigms and concepts (e.g. object-oriented with inheritance).

C is one of the most popular programming languages [4] and is one of the first languages taught to students. This fact also contributes to increase its popularity and it is largely used all over the world.

CTutor aims to be a widely used tool for program visualization. The common objective of this kind of tools is to help people better understand programs. These tools can be used by a beginner of programming or an expert and help users in the creation of mental models to easily associate the several elements of this language.

With this objective CTutor will be capable of fulfill a gap in the existing options to visualize programming languages in C, because the existing tools for this language are outdated.

Another goal of CTutor is to enhance the architecture of the program visualization tools.

The architecture of a typical program visualization tool uses an execution environment and a visualizer, and these two components are strongly coupled. Which means it is hard to separated them and reuse them to build other tools with same purpose. For example, if a professor needs a new visualizer for the program visualization tool, which he uses in his programming course, he will need to develop a new tool besides only a new visualizer.

A visualization tool that can reuse other visualizers, can present the elements of the language

in different manners. This reuse of components makes possible adaption to the student's or professor's needs. Also, changing the execution environment helps the student, which is familiarized with a specific visualizer, to use it with different programming languages.

CTutor will be based on an existing tool called Online Python Tutor [5]. This objective of CTutor is supported by a new architecture compared to the existing architectures in other tools.

The development of CTutor had two phases: first one was modifying Online Python Tutor's architecture, to split the visualizer and the execution environment; second phase was to develop a new programming tool to visualize C.

So, the first step of the implementation was splitting the visualizer and the execution environment into two separated components. This step started with the definition of the messages exchanged between those components, and the creation of a new component that will be the bridge of communication between them.

The second one was to analyze GDB/Machine Interface, to know how to interact with it and gather the information needed by the visualizer. This will help the development of the new execution environment that executes C programming language.

The following sections will introduce some concepts about a program visualization tool, such as the components, the target population, and the objectives of CTutor.

1.1. Visualizer

The visualizer is considered to be the part of the program visualization tool that shows to the student a graphic visualization of its own program. It is also responsible to know how to represent the different elements of a programming language.

1.2. Execution Environment

The execution environment is the part of the tool that executes the student's code. Additionally, it has to provide to the visualizer the several modifications that the elements suffer throughout the code. For example, if a variable change its value the execution environment has to expose this modification to the next component.

1.3. Target population

CTutor's users can be divided into three different entities, which have different goals when interacting with program visualization tools.

- The **student's** purpose is to use CTutor as a learning tool, using its visualizations to improve their knowledge about a certain programming language.
- The **professor** will use CTutor to teach a programming language, with the use of programming examples. With CTutor the teaching is more interactive than drawings on the blackboard or on slides.
- The **developer's** goal is extending or developing a new visualization tool. CTutor helps this actor with is new approach in terms of architecture of a program visualization tool.

1.4. Objectives

CTutor will be focused on program animation, which is the use of dynamic visualization techniques to enhance student understanding of the actual implementation of programs or data structures. This means that the student may examine the state of the program and related data, before and after execution of a particular line of code. To accomplish that CTutor defines two main objectives:

- As a program visualization tool, CTutor has the purpose of helping students of programming in learning or improving theirs programming concepts. A good analogy is a slow motion movie, because the source code of a program is transformed into several and sequential images of the operations or expressions represented in the code. The generated visualizations allows the student to analyze their code and understand precisely what the source code is doing in a certain line. Also, the student can assimilate the several components of a programming language and then create/improve his own mental models and reach the point where, in his mind, he can anticipate the next visualizations provided by the visualization tool.
- CTutor aims to facilitate the developing of new program visualization tools, which can be achieved with a different approach on the architecture of these tools. The developer can analyze the messages exchanged between the visualizer and the execution environment and then substitute one of them, or both. He can be concerned to develop only one component, and then the result will be a new tool, with different visualizations (in case of developing a new visualizer) or with a new language to be visualized (in case of developing a new execution environment). This fact will provide new program visualization tools and enable the customization of these tools with different visualizers to fulfill different requirements.

1.5. Document Structure

This document is structured in the following sections:

- **Chapter 2** exposes the related work. Presents two tools, one is a visual debugger, DDD, and

the second one is a program visualizatio tool, Online Python Tutor. Also it explains some inconveniences in the student's perspective.

- **Chapter 3 and 4** presents the three possible architectures to use as execution environment for CTutor, and how they can be integrated with a visualizer.
- **Chapter 5** presents the architecture and explains some details about the implementation of CTutor.
- **Chapter 6** performs the evaluation of the proposed solution. The evaluation is based on program examples taken from a book used as manual in programming course of C.
- **Chapter 7** concludes this document. It summarizes the developed work and introduces further work directions.

2. Related Work

This section summarizes the state of the art, although in this paper it is given more emphasis to DDD and Online Python Tutor. This tools are usually used in the in the context of learning a new programming language.

2.1. Visual Debugger

The Data Display Debugger (DDD) [6] is a graphical front-end for UNIX Debuggers. Despite the features of a debugger, like viewing source texts and breakpoints, DDD provides a graphical data display, where data structures are displayed as graphs.

DDD is one of the most used tools for C language, such as frontend debugger and for visualize the elements of the language.

The data window has several drawbacks for a beginner, the student has to explicitly open boxes to display information.

Another example is, if a student wants to visualize a certain behavior of the variable **x** located in *main* function, after he right-clicks it and chooses option **display**, he has another problem of visualization because this variable appear in data window without any link to the *main* function. Increasing the complexity of the problem, when a function f is call, for instance, from *main*, the environment (local variables) of the *main* function disappear.

In DDD function frames are not graphically represented, although they modify the way other data is represented.

This visualization features leads to a rejection from part of novice students and can also lead to flaws in the comprehension of simple things, such as definition of local scope of a function.

2.2. Online Python Tutor

Online Python Tutor [5] is a web-based program visualization tool for the programming language Python, developed by Philip J.Guo. Using this tool, a teacher or student can writes a Python program directly in the web browser and visualize what the computer is doing step-by-step as it executes the program.

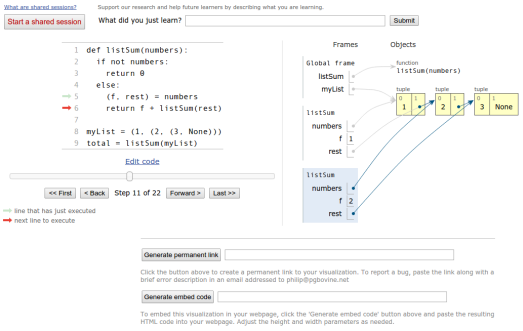

Figure 1: Online Python Tutor's user interface [5]

Important features of this tool are: is the only Python program visualization tool that runs within a web browser without any required software or plugin installation, and also is the only that can be easily embedded within webpages.

The backend is written in Python and takes the source code of a Python program as input and produces an execution trace as output. The input is executed under supervision of the standard Python debugger module, bdb [7], which stops execution after every executed line and records the program's run time state. Then an ordered list of execution points is created, where each point contains the state right before a line of code is about to execute, including:

- The line number of the line that is about to execute;
- The instruction type (ordinary single step, exception, function call, or function return);
- A map of global variable names to their current values at this execution point;
- An ordered list of stack frames, where each frame contains a map of local variable names to current values;
- The current state of the heap;
- The program's output up to this execution point.

After execution ends, the backend trace is encoded in JSON format [8], serializing Python data types into native JSON types with extra metadata tags.

Despite having a good visualizer , it is coupled to the execution environment. Also, Python Tutor is only web-based and it compiles all the source code before showing the visualization, so for each modification in the code Python Tutor will need to recompile and illustrate all the visualizations again. The visualizations are all made in the pre-execution, after that the visualizer has all the trace to do the

visualizations.

With this pre-execution mode, an inconvenience arises. For example, when a program needs a user input at a specific line (**raw_input**). The visualization goes normally until that line, and then an input box appears requesting for an input. After that, Online Python Tutor recompiles the program with this input, and starts over.

The explicit steps for this situation are:

1. The student inserts the source code in the frontend, and then is sent to the backend;
2. The backend executes the source code and sends to the frontend the program trace;
3. The student push forward in the visualization, and the frontend shows each visualization for each line;
4. When the next instruction needs the student input, the student inserts the input in a box;
5. The fronted sends all the source code again to the backend;
6. And the backend sends back a new program trace with the correct value for the raw_input;
7. The student begins the visualization from the start at this time there is no interruption in the raw_input.

2.2.1. Other Tutors

Based on Online Python Tutor, other tutors were developed, as Online Java Tutor [9], Online Ruby Tutor [10] and Online Javascript Tutor [11].

All Tutor's mentioned above are very similar to the Online Python Tutor in terms of visualization, but all of them are different in the backend and how they communicate with the visualizer.

The Online Java Tutor, uses *java_jail* [12] as backend, developed with the only purpose of creating the a Java version for Online Python Tutor. As a communication it is used a PHP script, with the intention of send the trace to the visualizer.

The Online Javascript Tutor, uses *Node.js* [13] as a backend and also uses it to send information to the visualizer.

The Online Ruby Tutor, is the one that has more differences in the visualizer, and as well in terms of backend. The backend is all in ruby.

These systems made major changes to be able to execute other languages. All these systems rebuild the backend in order of inspect each programming language, besides that they also changed the communication with the visualizer. The visualizer do not use the Google App Engine, to load it in the browser, instead the tutor's used other methods. Besides these modifications, these systems maintain the inconvenience existed in Online Python Tutor, all use a pre-execution state.

3. Execution Environment

In order to show the internals of a program, it is necessary to execute it in a specific environment. The execution environment will provide information, so that other component of CTutor produces the visualization for the student. These tools have different goals and functionalities, although they all have in common an important feature which is executing the source code and then, with some modifications, provide information about the several changes of the language elements (e.g. when variable changes its value).

3.1. Instrumentation

Instrumentation is a technique used to monitor an application while it is being executed. With instrumentation, programmers can insert additional code to the application, this code triggers messages with the state of the application.

To conclude, instrumentation is a way to gather information from source code, but for each modification on the code is needed a recompilation of the application.

3.2. Interpretation

PicoC [14] is a small C interpreter design for robotic, embedded and non-embedded applications, so it is multi-platform and portability.

PicoC has two ways of executing the source code, first one is interpreting the code and then presents the result and the second one is using a readeval-print loop (REPL). In a REPL the user enters an expression of C, then it is evaluated and the result displayed, after that the REPL waits another expression to begin again the loop.

PicoC as an interpreter parses the source code and perform its behavior directly, without previously compiled into machine code. In internal terms PicoC is composed by its own parser, lexical analyzers (lex), table and heap. The parser uses the lex to create tokens from the sequence of input characters. Then the parser will fill the table of symbols, which stores the identifiers, its values and types. In the heap is stored the memory allocation values, for example the values of a variable created with a *malloc* expression.

3.3. Debugging

A specialized debugging tool can significantly help to examine the dynamic behavior of a program. They provide support for inspecting the execution state in a symbolic way, and they allow for executing programs step-by-step or until a specific condition arises. Despite these functionalities, a graphical debugger also provides a display of the data structures, very useful for the user to follow and understand the execution flow of a program.

GDB, the GNU Debugger [15], came into exis-

tence around 1985, developed by Richard Stallman. In the UNIX world, it is one of the most popular debuggers and it provides a large range of functionality for all debugging purposes.

GDB provides two interfaces, command-line (CLI) and machine interface (GDB/MI [16]). Command-line interface basically uses the standard GNU library *readline* [17] to handle the interaction with the user. It is also able to do things like use cursor keys to go back in a line and fix a character.

One way to provide a debugging GUI is to use GDB as a backend to a graphical interface program, translating mouse clicks into commands and formatting print results into windows. But, it is not the ideal approach because sometimes results are formatted for human readability, omitting details and relying on human ability to supply context.

To handle this problem exists the second interface provided, machine interface. It has the following features: Commands and results have additional syntax that makes everything explicit - each argument is bounded by quotes, and complex output has delimiters for subgroups and parameter names for component pieces. Also, commands can be prefixed with sequence identifiers that are echoed back in results, ensuring reported results are matched up with the right commands.

4. Integration with Visualizer

This section presents how to accomplish the integration with the visualizer.

Figure 2 represents three possible architectures used for the developing a program visualization tool. These three architectures have as a starting point the source code, produced by the student.

1. The use of instrumentation, requires the modification of the source code. After that the instrumented application executes the modified source code. The visualizer 1 is responsible for create the trace and to control the execution. This trace is send to the visualizer 2.
2. Alteration of the execution environment enables the possibility of the environment to execute the source code and gather the information to send for the visualizer 1+2. For example, examining the parse tree to know when its created a new variable.
3. After the compilation of the source code, it is possible to use an execution environment to inspect the application. This execution environment will automatically create triggers that can be exploit with an interface. This interface is used by the visualizer 1 to gather the information and send it to the visualizer 2.

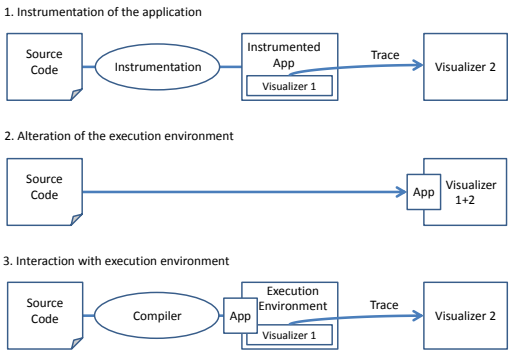

Figure 2: The three different methods to build a program visualization tool

Table 1 presents those methods. Independently on the architecture, there are several ways to collect information.

Execution environments	**Advantage**	**Disadvantage**
Instrumentation	Performance	Too complex Needs recompilation
Interpretation	Adaptable to the browser Used to develop an interactive tool	Only supports a subset of the language Change language implies changing interpreter
Debugging	Support several languages Easy access to application state Automatic code modification (gcc -g)	

Table 1: Comparison between solutions to Execution Environment

5. CTutor

This chapter will explain the choices made in the architecture of CTutor, i.e which execution environment will be used, and which visualizer. The decisions are made with the purpose of accomplish the objectives of CTutor, being a program visualization tool for C, and facilitate the creation of new tools with existing visualizers or existing execution environments.

The second section will present the components of CTutor and their implementation, as well the interactions between them.

5.1. Architecture

The Figure 3 is a modified version of the Figure 2, and it explains the future role of CTutor in each of the three architectures, and also where it will be necessary CTutor to be implemented.

1. Using instrumentation, will require changes the source code and as illustrated CTutor will have direct contact to it, then it will communicate to the visualizer the different state changes;
2. The interpreter will be the responsible for reading the code. Then, CTutor will change the interpreter to capture that information and send it to the visualizer;
3. When using a debugger, such as GDB, it will analyze the code and then it will provide to CTutor information in its own format, in case of GDB it is an GDB/MI. CTutor will need to know this format and how to interpret it, having then the information and transmit it to the visualizer.

To use instrumentation as backend to CTutor, a modification in GNU Compiler Collection (GCC) [18], would be a complicated task.

The interpretation, for instance with PicoC, needs to transverse the parse tree, and for that it will be needed to learn the code of this interpreter. Also it will be needed its modification, not only for intercept the cases for the visualization, as well for communicate with the visualizer.

The use of GDB, allows the use of a generic backend without modifications on it, CTutor only the need to interact with it. This interaction is done through its GDB/Machine interface, presented in the section 3. Another interest of using GDB is the possibility, with slightly changes, of using other programming languages without the inconvenience of construct of a new backend.

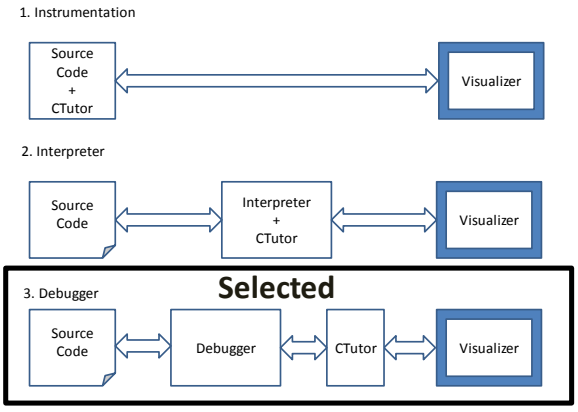

Figure 3: Preliminar hypothesis to CTutor's architecure

The Visualizer used in CTutor is the same as the Online Python Tutor. This visualizer has been used in some programming courses, in universities as well as in e-learning. It has also some features, that can be handy to the learning process, the possibility of embed in a webpage the visualization, and sharing live visualization sessions.

The resulting architecture of CTutor is represetend in the Figure 4.

5.2. Implementation

The implementation of CTutor starts in the separation of the execution environment and the visualizer of Online Python Tutor (OPT). This separation between the visualizer and the execution environment, enables the reuse of the visualizer of OPT.

Compared to OPT, CTutor has a different execution environment code and the TutorStub to link both parts. As shown in Figure 4.

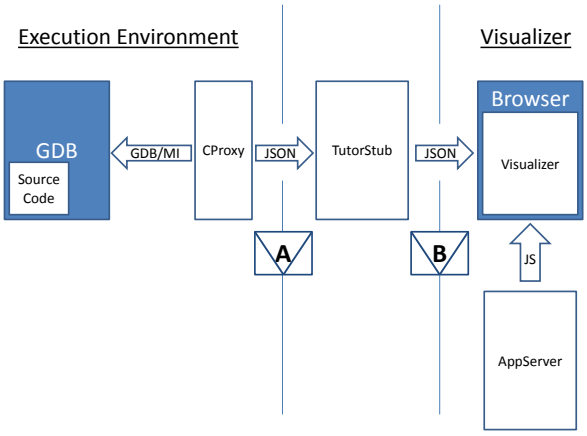

Figure 4: The architecture of CTutor

The execution environment is made by the GDB and a component called, CProxy. The CProxy handles the interaction with GDB, through an existing library, and the communication with TutorStub, throughout JSON objects,.

The CProxy is implemented in the file **cproxy.c**. Since the communication, with TutorStub, is done by JSON objects. CProxy uses a library that maps C **structs** in that format. This library is cJSON [19], portable, single-file, ANSI-C compliant JSON parser.

The GDB/Machine Interface library (libmigdb) [20] implements the GDB/MI protocol, allowing CProxy to controll the application and retrieve its state.

It was necessary to add two methods to libmigdb: **gmi_break_insert_main** and **gmi_stack_lis_variables**. The first one is used to create a breakpoint into main function, as is the only entry point of C programming language,

so that all visualizations starts always in main function. The **gmi_stack_list_variables** is used to gather the local variables and function arguments for the selected frame.

For each line, after the breakpoint, CProxy verifies the stack of frames (**gmi_stack_list_frames**) and the stack of variables (**gmi_stack_list_variables**). The outputs of these two methods are processed with the intention of getting information to fill the cJSON struct, for example name and value of a certain variable.

The TutorStub is a HTTP-Server, based on the python web framework Bottle [21]. This server provides two services *getSourceCode* and *getNextInstr*.

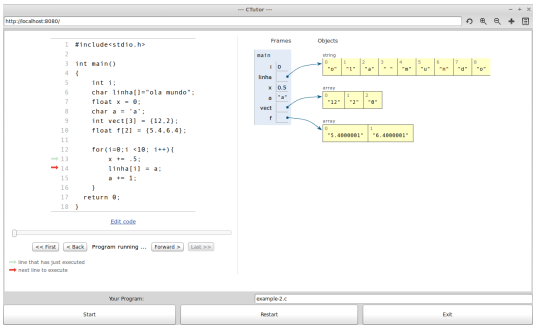

Figure 5: User Interface of CTutor

Comparing to OPT, CTutor changes the way students inserts source code to run. OPT, which uses a code editor on the browser to get student's code, while the CTutor gets the script from a file on the hard drive. The *getSourceCode* is invoqued from the javascript code on the visualizer.

The *getNextInstr* is used by the browser to request the next instruction trace, thus it can show the visualization. This request is sent when the student press the button **forward** on the visualizer.

The program trace is a list of JSON objects, where each object correspond to a single instruction trace and it represents a message with the changes that the visualized environment suffers over the time. This changes are declaration of variables, modification of its values and execution of functions.

The TutorStub reduces the existing coupling between backend and frontend, working as a proxy. The TutorStub receives an instruction trace (A) from the CProxy and send to the browser the program trace (B), that are represented as envelopes in Figure 4.

The appserver is a Google App Engine that allows the execution of Python code, using a pre-loaded Python interpreter in a safe "sandboxed" environment, used in OPT to execute the python debugger in the browser and to send the trace to the visualizer. In CTutor, as the execution environment uses the GDB and is not executed in the browser, also the TutorStub takes the responsibility of sending the trace to the visualizer, this server only loads all the contents of the visualizer (e.g. html, js and css files) to the browser.

There are two ways of launch CTutor, one is using a python script **main.py** and the other is using a graphical user interface (GUI) **gui.py**. As CTutor is a tool for students the gui is more suitable, but for developers the main script can be more useful to understand how to launch the components.

The main script is used to launch the two servers, the browser and for the user specify which file he wants to analyze in the visualizer (browser).

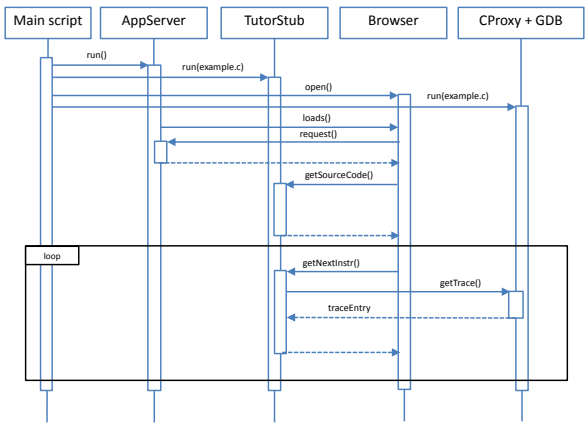

Figure 6: Interactions between components of CTutor with Browser

This GUI, presented in Figure 5, was developed using the *pywebkitgtk* [22], this package enables the embedding the a web-page into the GUI. Another feature is about with the selection of the program, it as an input text box which is checked in the start of the server to get the file name of the file that the student wants to visualize.

The Figure 6 and 7 are sequence diagram to explain the interactions between the components of CTutor, also they demonstrate the difference between the CTutor using the main script and using the GUI.

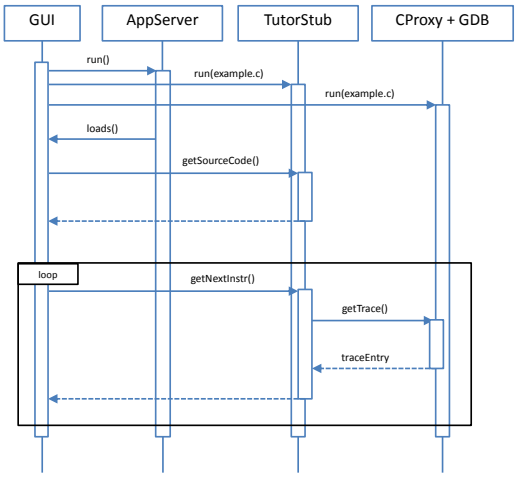

Figure 7: Interactions between components of CTutor with GUI

Observing the Figure 4, that shows the architecture of CTutor, it is possible to identify in the sequence diagrams the division between visualizer and execution environment. There are no direct interactions between the visualizer (Browser) and the execution environment (GDB). The TutorStub and the CProxy between them, reduce the coupling.

The loop represents the visualization process, as the student is visualizing each instruction it will send a *getNextInstr* request to the browser (whit button forward on the visualizer), passing through the TutorStub and transformed into a *getTrace* request to the CProxy. After that the reverse path, start with the execution of the instruction in GDB, the CProxy interacts with it to know which elements of the language were change, builds the trace and then send it to the TutorStub as a traceEntry (piece of trace, called instruction trace).

The biggest difference between the use of a Main script or a GUI, is the presentation to the student. As the GUI embeds the browser and occults the use of a console to do the start of the other components it is more suitable for students. The Main script can be more useful for developers to analyze the traces exchanged between the components.

6. Solution Evaluation

The evaluation of CTutor was based on the programming examples of the book "Linguagem C" [23]. This book is usually the manual of C in first courses.

The practicability of using a programming visualization tool as a complement of the learning process was demonstrated using the programs presented in the book. With this it is possible to evaluate CTutor.

The Table 2 presents the elements of the C language supported by CTutor. These elements are divided in groups following the book's chapter.

Elements of language	
Basic Data Types (char, int, float and double)	✓
Operator Types	✓
Constants	✓
Control Statements (if..else, switch, loops)	✓
Functions	✓
Arrays	✓
Multidimensional Arrays	✓
Strings	✓
Input and Output (printf, scanf)	✓
Pointers	+/-
Command Line Arguments (argc, argv)	✓
Files	✓
Structured Datatypes	++/-
Dynamic Memory Allocation	X
Pre-Processors and Macros (#include, #define)	✓
Multifile program	✓

Table 2: CTutor's supported elements of language C

The first conclusion reached with the information presented in the Table 2 is that CTutor supports all the C language. But, there are some elements that are not visualized, this means that CTutor runs the program examples, yet the visualization is not visible as expected.

For example, in the chapter about Dynamic Memory Allocation, it would be interesting when a student uses the function **malloc**, the visualization demonstrates the memory reserved, instead of hiding this important concept.

As CTutor uses an adapted visualizer from Online Python Tutor and the concepts of some elements in C are hard to divide, some issues occur on the visualizations.

The visualization of multidimensional arrays is not the expected, instead of a matrix represented by a table, it is shown a graphic that represents a variable with type **int*[]**. In Figure 8 it is represented a matrix of type integers.

Figure 8: Example of a multidimensional array

The value of a pointer in C, is a memory address, but this address can belong to other elements so

how to represent this in suitable way. In order to facilitate the visualization for the student, a pointer is presented in the visualizer with its address when it is created, when its address is the same of basic data types, Figure 9. The pointer is represented as an arrow to other elements, when its address is the same as a complex data types (string or array), Figure 10.

Figure 9: Example of a pointer to a char

Figure 10: Example of a pointer to an array of chars

Usually the structs in C are dependent on pointers, for example the use of lists in C. As seen in Figure 11, the visualization of this element can be misunderstood, because the visualizations of a struct and a pointer to it are the same. Ideally, it is better to change this issue, one way to solve it is using a brace "{" to associate the struct to the variable created in the frame main.

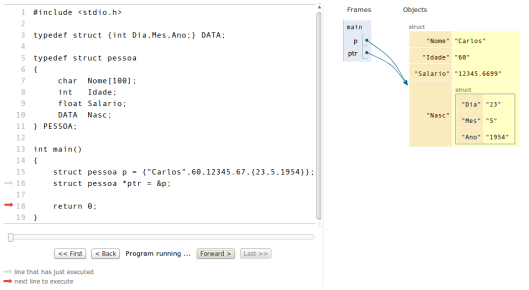

Figure 11: Example of a struct and a pointer to it

7. Conclusions

Programming is a complex domain in which students need a constructive support from their learning environments, to help students to understand programming, with this purpose some tools such as program visualization tools exists.

This program visualization tools can be a great aid for the usual learning process of a new programming language, being a complement of the classes or the e-learning courses. Also, if a student has the need to learn by himself this kind of tools provide an opportunity to learn by simple examples.

After doing some research on existing program visualization tools, it is possible to identify the three components of these tools: the execution environment, the visualizer and the language supported. The problem in the existing tools is that we cannot split the first and second components and reuse it in another tool. For a novice student, which is already attached to a particular visualizer, it will be difficult to change when he needs to learn another language, he will forced to use another tool.

The professor can also have some problems when switching between tools for teach each language, for example the examples created can be better visualized in certain visualizers and thus prevent the best explanation for the student.

CTutor solves this problem with a new architecture. The architecture of CTutor is constituted with a visualizer and an execution environment but separated in diferent processes and with a well defined interface.

The implementation of a new component, TutorStub, between the execution environment and the visualizer leads to a reduced coupling between them. The developing of a CProxy to interact with the execution environment, in this case GDB with its GDB/ Machine Interface, enables the construction of a trace for every execution point and outside of the GDB, avoiding its modification.

This architecture can facilitate the development of new tools and also a mix between visualizers and execution environments. With a minor effort, comparing of building a tool from scratch, the developer can implement a component adjusted to his needs.

The evaluation of this solution was made using programming examples from a book, the result is that CTutor can support most elements from the C language but has some issues in the visualization of pointers.

After all, CTutor is a program visualization tool that can visualize almost all the examples used to teach C, as well its architecture can facilitate the developing of other visualization tools, visualizers and execution environments separately.

7.1. Future Work

In the future, developers have multiple ways to proceed the work done until here, such as:

- Incorporate CTutor in different learning environments or to support other languages;
- Enhance the portability of CTutor, porting it to the web;
- Improve visualizations of structs, dynamic memory and pointers;

References

[1] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. Prentice Hall Professional Technical Reference, 2nd edition, 1988.

[2] Eclipse ide. `http://www.eclipse.org/`. accessed 20.11.2014.

[3] Netbeans ide. `http://www.netbeans.org/`. accessed 20.11.2014.

[4] Programming language popularity. `http://www.langpop.com/`. accessed 20.11.2014.

[5] Philip J. Guo. Online python tutor: Embeddable web-based program visualization for cs education. In *Proceeding of the 44th ACM Technical Symposium on Computer Science Education*, SIGCSE '13, pages 579–584, New York, NY, USA, 2013. ACM.

[6] Andreas Zeller and D Lütkehaus. DDDa free graphical front-end for UNIX debuggers. *ACM Sigplan Notices*, 1996.

[7] bdb - debugger framework. `https://docs.python.org/2/library/bdb.html`. accessed 20.11.2014.

[8] Introducing json. `http://json.org/`. accessed 20.11.2014.

[9] Online java tutor. `http://cscircles.cemc.uwaterloo.ca/java_visualize/`. accessed 20.11.2014.

[10] Online ruby tutor. `http://www.basicruby.com/tutor`. accessed 20.11.2014.

[11] Online javascript tutor. `http://jstutor.herokuapp.com/`. accessed 20.11.2014.

[12] Java jail. `https://github.com/daveagp/java_jail`. accessed 20.11.2014.

[13] Node.js. `http://nodejs.org/`. accessed 20.11.2014.

[14] Picoc: A very small c interpreter. `https://code.google.com/p/picoc/`. accessed 20.11.2014.

[15] R.M. Stallman and R.H. Pesch. *Debugging with GDB: The GNU Source-level Debugger*. Free Software Foundation, 1992.

[16] The gdb/mi interface. `https://sourceware.org/gdb/onlinedocs/gdb/GDB_002fMI.html#GDB_002fMI`. accessed 20.11.2014.

[17] The gnu readline library. `http://cnswww.cns.cwru.edu/php/chet/readline/readline.html`. accessed 20.11.2014.

[18] Gcc - the gnu compiler collection. `https://gcc.gnu.org/`. accessed 20.11.2014.

[19] cjson. `http://cjson.sourceforge.net`. accessed 20.11.2014.

[20] Gdb/machine interface library. `http://libmigdb.sourceforge.net`. accessed 20.11.2014.

[21] Bottle: Python web framework. `http://bottlepy.org/docs/dev/index.html`. accessed 20.11.2014.

[22] Python bindings for the webkit gtk+ port. `https://code.google.com/p/pywebkitgtk/`. accessed 20.11.2014.

[23] Luís Damas. *Linguagem C*. FCA - Editora Informtica, 17^{nd} edition, 1999. ISBN:978-9727221561.