Adam B. Mtaho
PhD in Computer Science, Lecturer of Information and Communication Technology Department
Arusha Technical College, Arusha, Tanzania
ORCID ID 0000-0002-6997-3332
abasigie@yahoo.com

Masoud M. Masoud
PhD in Computer Science, Deputy Director, ICT & Technology Development Department
Tanzania Industrial Research and Development Organization, Dar es Salaam, Tanzania,
bigutu@gmail.com

Leonard J. Mselle
PhD in Computer Science, Associate Professor of Computer Science and Engineering Department
University of Dodoma, Dodoma, Tanzania
mselel@yahoo.com

DEVELOPING A CELIOTM PROGRAMMING LEARNING TOOL TO FACILIATE TEACHING AND LEARNING DATA STRUCTURE CONCEPTS IN C++ FOR NOVICE PROGRAMMERS

Abstract. Programming is the core skill in computer science (CS) education. It is also a useful course in engineering and science-based courses. However, teaching and learning computer programming has not been an easy task. This is evidenced by the fact that majority of the students face challenges and difficulties in understanding programming concepts and how to apply them in real-life scenarios. The situation is worse for the Data Structures and algorithms (DSA) course, an advanced-level programming course that is mandatory for any CS student. The subject is too hard for novices to grasp due to its abstract and dynamic nature. To address such difficulties, several algorithm visualization (AV) tools have been introduced to help novices understand data structure. However, the pedagogical effectiveness of using such tools has not been successful because they are less engaging for learning DSA. This paper describes how the CeliotM programming learning tool was developed to facilitate learning data structure concepts in C++. The development of CeliotM was achieved by using reuse-oriented software engineering approach. CeliotM was developed by redesigning a Celiot program visualization (PV) tool-a programing learning tool that supporting learning programming in C++. Thus, by using Java as the language, the original version of Celiot was resigned to support compilation and visualization of various data structure elements in Memory Transfer Language (MTL) format; and incorporate several learner engagement features, including the inbuilt C++ compiler and animation explanations. The resulting tool is a CeliotM programming learning tool that visualizes and compiles data structure objects such as queues, stacks, and linked lists in C++ programming language. Empirical results on the evaluation of using CeliotM in teaching data structures and algorithmic concepts reveal that using such tool enhanced students' programming comprehension and offered a more appealing learning experience for novice programmers. The greatest contribution of this work is to provide an education tool for teaching data structures in C++ that can work as a compiler, program and algorithm visualization tool in tandem. It also contributes a valuable resource to programming education, offering an effective and inspiring approach for novices to grasp fundamental programming and data structure concepts.

Keywords: CeliotM; Data structures Visualizations; MTL; Learning Programming

1. INTRODUCTION

Programming is the core skill in Computer Science (CS) education. It is also a useful course in engineering and science-based courses. However, teaching and learning computer programming has not been an easy task[1][2]. Data Structures and Algorithms (DSA), is the advanced course in programming after introductory programming. DSA is mandatory for any CS student. This course is purported to entail a higher cognitive load compared to introductory programming courses as it demands learning more abstract data types and various algorithms [3][2][4]. A survey by [2] showed that 56% of students' grades were lower in the DSA course than in the introductory programming course for more than three hundred undergraduate students who were enrolled at the University of North Carolina, USA. The high failure rate in learning DSA is a significant factor causing many students to lose motivation in pursuing programming courses in colleges and universities worldwide.

In learning DSA, majority of novices fail to to master the basic principles for effective program design and implementation. They also fail to comprehend the commonly used data structures, their related algorithms, and most of the design patterns common in programming [3] [5].

Computer codes have both static and dynamic characteristics. The plain text on the editor represents the static part while its compilation and execution represents the dynamic part. To all novices the dynamic character of the code is unknown until when this is compiled and executed. Since compilation and execution are machine driven, the dynamic nature of the code remain mysterious to most novices. To help novices understand the dynamic execution of the computer programs and algorithms, algorithm visualizations (AV) and program visualization (PV) tools were introduced [6] [7]. AV tools are used for teaching data structure and algorthms concepts in CS2 course while PV tools are used for teaching basic programming concepts in CS1 course. AVs are used for visualizing data structures and their operations according to a particular algorithm and the transition between those states during program execution. PV, on the other hand, highlights the code and displays the data on a variable to visually represent how the program is being executed. However, the use of AVs in teaching DSA has not been successful as expected because they are less engaging for learning DSA [8][6].

PROBLEM STATEMENT

Currently, the DSA course in many universities is still instructed by using the traditional lecture method, along with a few AV tools [9]. Some of these AV are incompatible with DSA textbooks used for teaching [10], [9]. Besides, AV tools, unlike PVs, have focused mainly on demonstrating algorithm steps instead of showing the programming logic behind those steps [11] [12]. The majority of existing AV tools function as machine-driven entities, lacking integration with standard compilers and compatibility with learner-driven tools [13][12][10][14], resulting in extraneous cognitive load (ECL) in learning programming.

The research goal

This study aimed to develop a new programming learning tool (CeliotM) to support learning data structures concepts in C++. The tool was expected to work as a PV, AV, and compiler.

Analysis of recent studies and publications

There exist several PV/ AV tools that are used to assist learning programming. This section reviews four AV/PV tools that are mostly related to this study. Such tools are Jeliot3 [15], Courseware[16], PITON [17], and Celiot[13].

Jeliot 3

Jeliot 3 [15] is a PV designed to support novices learning of both imperative and object-oriented programming paradigms in Java. The key feature of Jeliot 3 is the full or semi-automatic visualization of the data and control flows. Jeliot 3 is published as an open-source project under General Public Licence (GPL) and is available for free from the web. Jeliot 3 has been evaluated in numerous studies with results showing that its use enhances students' comprehension. However, in other studies, results were negative [18]). Despite the widespread use of Jeliot 3, the tool lacks informative error messages [13]. Additionally, it is not integrated with the compiler, resulting in switching overhead when using the tool.

Courseware

The Courseware or Data Structure Courseware [16] is a computerized system which incorporates both a PV and AV. It was developed as an AV/PVtool to aid students in studying DSA and enhance students' understanding. The Courseware supports the learning of a wide range of DSA topics such as sorting, searching, linked-lists, stack, queues, trees, and binary search trees. The tool works both as AV and PV. Results from evaluation of Courseware showed that students who used Courseware performed better than those who were instructed using the traditional lecture method. One major drawback of the Courseware is its lack of a user-friendly interface, making it difficult for slow learners to use it. Additionally, the tool heavily relies on pre-defined, randomly generated values for sorting and searching algorithms, limiting learners' flexibility to test their algorithms with their datasets and codes.

PITON

PITON [17] is a program animator developed to assist novices learning introductory programming in Python language. PITON combines both PV and the programming working environment, thus helps reduce switching overheads in using the tool. Results from the PITON evaluation indicate that the tool is useful in learning programming. However, PITON suffers from usability and pedagogical deficiencies due to limited PV features, hindering the development of programming solutions from scratch.

Celiot

Masoud [13] developed Celiot to assist instructors and students in teaching and learning introductory programming courses in C++. Celiot tool was developed as an attempt to overcome the shortage of PV tools that support learning programing concepts in C++. Celiot was created by combining Memory Transfer Language (MTL) [19] and Jeliot 3 [13]. Celiot employs the design combination of Jeliot 3 which is a machine-driven animator and the design of MTL which is a learner-driven debugger [13]. Celiot has been evaluated in learning introductory programming courses both at university and college levels. Results show that the use of Celiot improves novice programming skills for students studying basic programming in C++ [13]. Despite the pedagogical effectiveness of using Celiot, still it had several limitations. Celiot supports only a relatively small subset of C++ language basic programming concepts. It does not support data structure concepts. Celiot provides limited support for pointer visualization since it lack all functionalities to handle all programming issues relating to a pointer and dynamic memory allocation. It also lacks compiler and does not augment animations with text explanations.

Several researchers have made numerous efforts to design and implement educational support tools for learning programming. The reviewed studies and tools indicate that deficiencies in the majority of existing PV/AV tools tend to diminish the overall efficiency of using visualizations in teaching and learning programming courses. This is because their usage often adds a high extraneous cognitive load to learners instead of minimizing it. More emphasis on developing tools that are more learner-engaged has been given by several researchers, such as [8], [20], and [12]. They emphasize the need to develop programming learning environments that combine the use of AV, PV, compiler, and tutorials in one package. Likewise the studies by [13]and [21] report that there exist a limited number of PV/AV tools that visualize programs in C++, and most of them are inactive or outdated. To address this deficiency, researchers have emphasized using programming learning supporting tools that provide both learning and working environment [13][22][16].

2. METHODOLOGY

To develop CeliotM, the reuse-oriented software engineering approach [23] was used. Software reuse is simply the act of using existing software resources and knowledge, such as code, designs, and whole components to create a new software product. Figure 1 shows the steps of reuse-oriented software engineering approach adapted from [23].

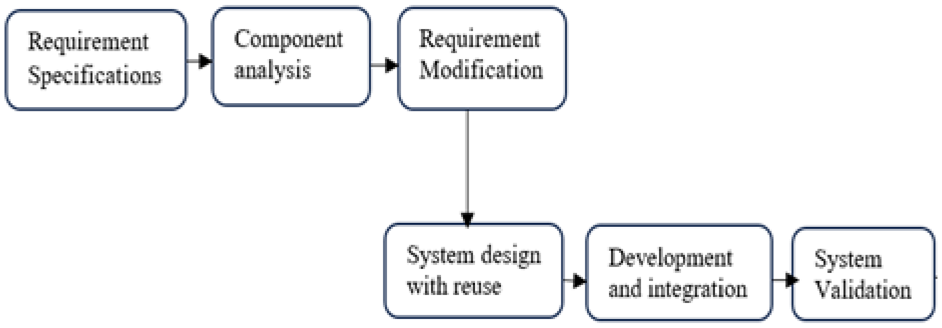


Figure 1. Reuse-oriented software engineering approach adapted from [24]

The software that was selected for re-use in developing CeliotM was Celiot [13] and Standard C++ compiler. Celiot was selected due to the fact that it met the requirements specified in the study and all criteria for reuse as specified by Somerville [23]. According to Somerville [23], such criteria are: the availability of source code; the presence or absence of support; the possibility of using a component library; and the ease with which to find, understand, adapt, and give reusable components to work in a new environment. The Java programming language and Net Beans integrated development environment (IDE) were software used during coding. During the development and integration process, the new features that support data structures objects were developed by modifying the Celiot architecture through laboratory works. Data structures, pointer, and memory addressing and other pedagogical supporting features were embedded and integrated with existing Celiot features to produce CeliotM through the re-designing and re-programming of the Celiot visualization engine. The resulting CeliotM prototype was tested in the laboratory and refined for improvements before being deployed for use in actual class settings for empirical evaluations.

Requirements for Development of CeliotM

In order to determine suggestions (requirements and guidelines) for designing and implementing CeliotM framework, this study used literature review, workshop, and interview methods. Some of the suggestions found in literature concerning the need of developing the programming learning support tool to overcome difficulties in teaching and learning DSA includes those of Naps et al. [25], Vagianou [10], and [12]. Naps et al. [25]suggested complementing animation within text or verbal explanations. Masoud [13]recommended having a tool that combines the use of both learner and machine-driven-functionalities for teaching DSA, and that works as AV and PV. Vagianou [10], also recommended integrating learner-driven with machine-driven animated tools, [12] recommended combining PV with

IDE. Table 1 summerizes nine (9) system requirements (features) used in developing the CeliotM programming learning tool.

Table 1

System Requirements for Developing CeliotM Framework

S/N	Requirements	I	S	L	R
1	The proposed system shall enable students to compile and visualize DSA programs in C++.	√	√	√	
2	The proposed system shall enable students to visualize data structure elements in both imperative and object programs in C++.	√	√	√	
3	The proposed system shall incorporate animations with both user and system-defined explanations.	√	√	√	√
4	The proposed system shall be used along with congruent learner-driven, MTL-based instructional materials.	√	√	√	
5	The proposed system shall display the memory address during the animation.	√	√	√	
6	The proposed system shall have an enhanced user interface.	√	√	√	
7	The proposed system shall work as an AV, PV, and compiler in one package.		√		√
8	The system shall allow zooming in and zooming out of the visualized objects.				√
9	The system shall provide enhanced tutorial modules within the tools.		√	√	√

Key: I= Instructors’ views ; S=Students views, L=Literature, R= Researcher and Experts´ observations

General design and Architectures of the software to be reused-Celiot

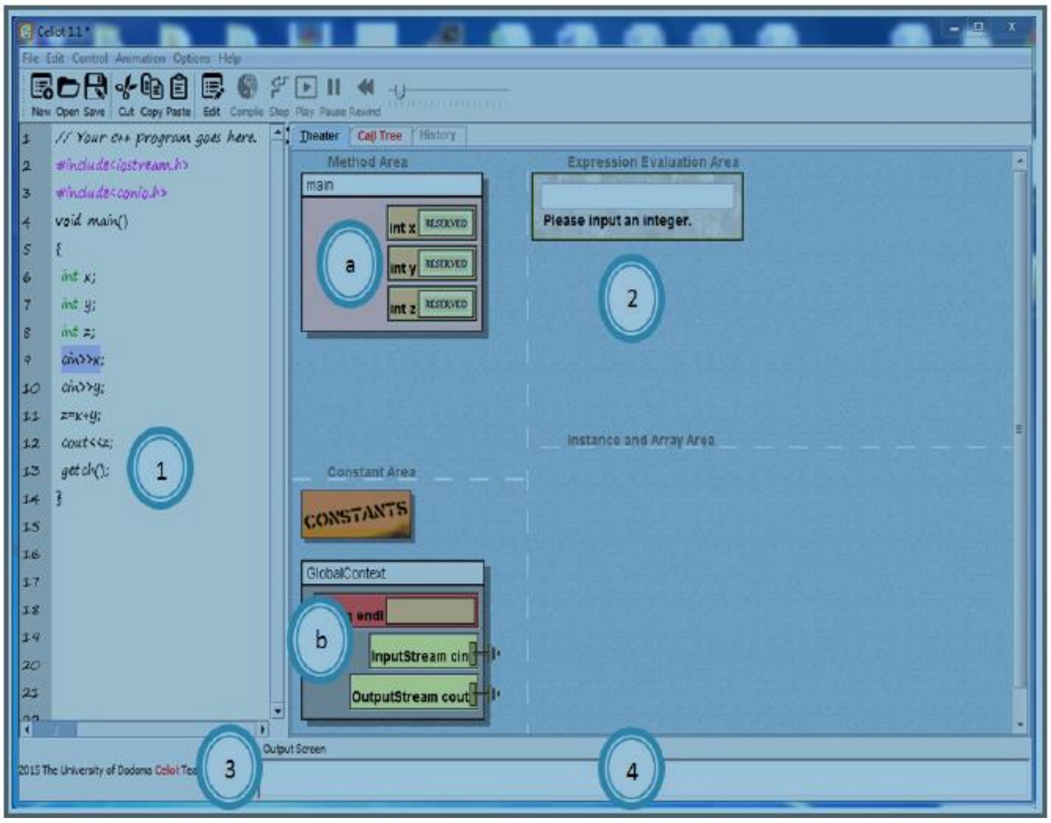
As stated earlier, the aim of this study was to develop a programming tool that could work as a PV, AV, and compiler. Since the study used the reuse software engineering model as a software development methodology, after determining the requirements, it was necessary to analyze which software would be selected for reuse. Thus, in this study, Celiot and a C++ command line compiler were used to develop CeliotM. Celiot was selected as the main software to be reused due to the fact that it met the requirements specified in the study as defined by Somerville [23].That is, because Celiot is open source and falls under a public license, its source codes can freely be made available for software reuse. The tool source code provided a room for reusing a component library for developing the proposed system. The Celiot source codes were easy to find, learn, understand, and adapt, making software reuse possible. Moreover, since the Celiot was designed under the MTL framework, the tool was chosen because it could be extended to support MTL for data structures. The C++ compiler that was integrated into CeliotM is a free small-sized command-line compiler that ships with the C++ IDE. The compiler was extracted from IDE by installing IDE (on a chosen Windows PC) with minimal installation options.

Celiot Design

Celiot is a member of Jeliot 3 family [13]. It is a program animation tool designed to assist instructors and students in teaching and learning introductory programming courses in C++. This tool was developed as an attempt to overcome the shortage of PV tools that support learning basic programming concepts in C++. Celiot was created by combining Memory Transfer Language (MTL) and Jeliot 3 [13]. Celiot employs the design combination of Jeliot 3 which is a machine-driven animator and the design of MTL which is a learner-driven debugger [13]. While Jeliot 3 visualizes Java programs [26], Celiot visualizes basic C++ programming concepts such as int, char and string data types, control of structures , iterations, arrays, and functions. It also visualizes programming operations such as variable declaration, data feeding, assignment, data processing, data storage, and outputting [13]. Celiot runs under Window Operating systems (all versions of window). Due to the fact that it is written in Java, Celiot needs Java Development Kit (JDK) to be installed in the computer to enable it to work. To enable one to use Celiot, its installation package must be installed in a computer and if a computer does not have the JDK software installed in it [13].

Celiot User Interface

Figure 2 shows the user interface of Celiot. As depicted in Figure 2, Celiot consists of four main panes namely the code editor (1), visualization pane (2), company's log (3) and output console (4). The code editor contains the codes to be visualized. The visualization panel contains the four interconnected areas namely; the method, expression evaluation, constant, and instance and array areas for different components of visualization. The output console is located at the bottom of the frame next to the company log. In the method area, MTL diagram labeled (a) always visualizes the dynamic behavior of the variable. The library calling visualization always lists some functions which reside in that library as depicted in the constant area labeled (b) as noted by Moreno and Myller [15][27].


Celiot 1.1 *
File Edit Control Animation Options Help
New Open Save Cut Copy Paste Edit Compile Step Play Pause Rewind
Theater Call Tree History
Method Area
main
int x RESERVED
int y RESERVED
int z RESERVED
a
Expression Evaluation Area
2
Instance and Array Area
Constant Area
CONSTANTS
GlobalContext
endl
InputStream cin
OutputStream cout
b
1
3
Output Screen
4
2015 The University of Dodoma Celiot Team

```
// Your c++ program goes here.
#include<iostream.h>
#include<conio.h>
void main()
{
 int x;
 int y;
 int z;
 cin>>x;
 cin>>y;
 z=x+y;
 cout<<z;
 getch();
}
```

Please input an integer.

Figure 2. User interface of Celiot adopted from Jeliot 3 Tree [15]

Celiot Architecture

The Celiot architecture resulted from the modification of the Jeliot 3. Jeliot 3 by its structure is incapable of animating C++ programs. Celiot on the other hand can visualize basic programming concepts in C++. However, it does not visualization and compile data structure programs. According to Masoud [13], Celiot's initial design was derived from Jeliot 3 through the incorporation of MTL features. The first aim of developing Celiot was to make Jeliot 3, capable of animating C++ programs. Figure 3 shows the general design of Celiot.

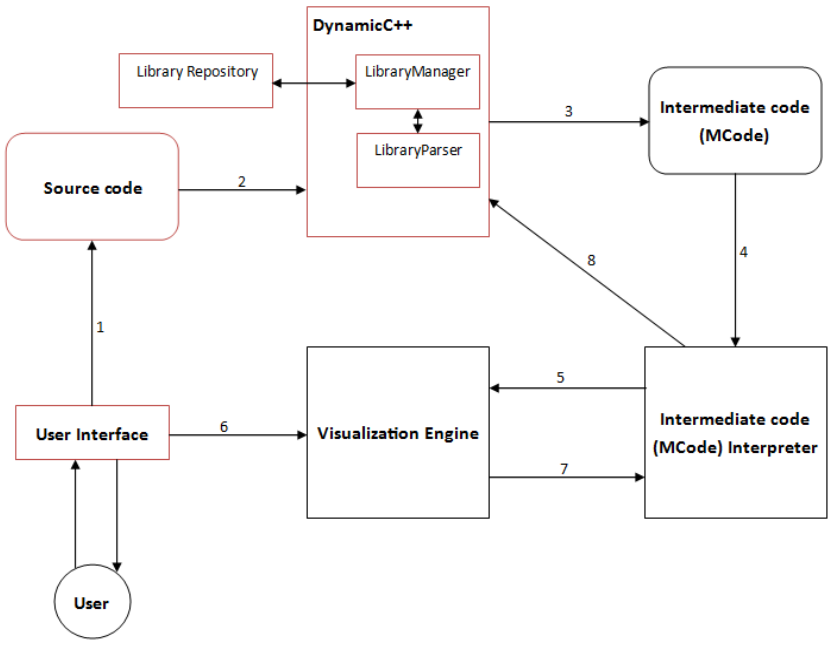


Figure 3. Celiot Design Architecture Masoud [13],

As shown in Figure 3, in Celiot, the code viewer and animation engine are separated by the Dynamic C++. The Dynamic C++ is a tool used to connect the visualization engine and the code viewer. When users write their codes in the editor, the code is taken into the DynamicC++ for interpretation. After interpretation, the DynamicC++ produces the Mcode which is used by the visualization engine to produce the image in the form of RAM diagrams[19]. The MCode carries the semantic of the interpreted program. The modifications that were done in Jeliot 3 to produce Celiot only enabled the tool to visualize fundamental C++ programming features but left data structures and pointer untouched. Figure 4 shows the structure of Dynamic C++.

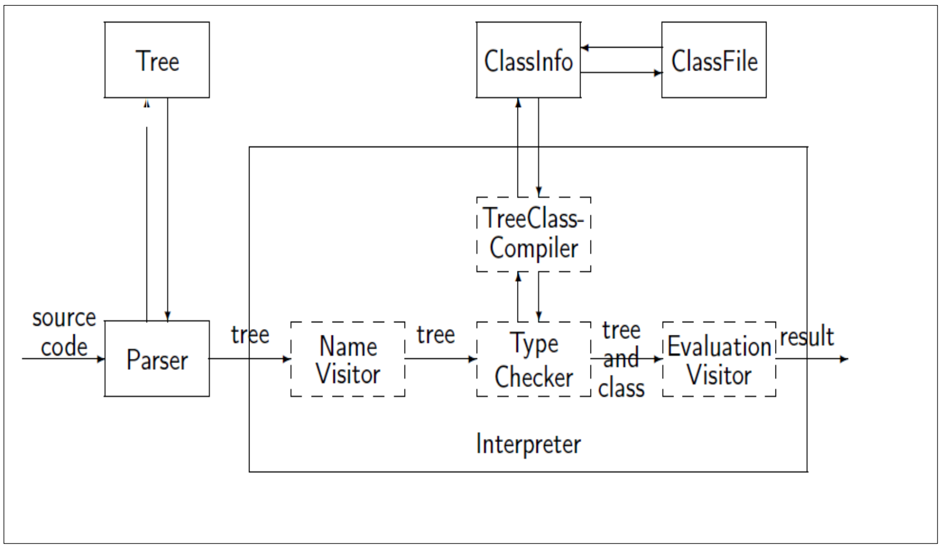


Figure 4. The structure of DynamicC++

The Dynamic C++ consists of two main parts, the Parser which performs syntactic analysis on the program producing the syntax tree; and the Interpreter which extracts the semantic of the program from the syntax tree and produces the intermediate code, MCode. The Parser accepts programs' source codes, divides it into tokens and performs syntactic check on them. While checking the tokens, if a series of tokens is found to construct a syntactically correct part of the program, the parser creates a node for the tokens filling important info in the Node and inserts the Node in the syntax tree representing the program. The Interpreter accepts the syntax tree and traverses the nodes in it to execute and extract the semantic of the program. While the Nodes are traversed, they are passed through three components which are NameVisitor, TypeChecker and Evaluation Visitor.

The NameVisitor is a tree visitor that resolves the ambiguity in identifiers in a syntax tree. This visitor traverses the tree trying to find out syntactical ambiguities. The Type Checker is a tree visitor that checks the typing rules and loads the classes, fields and methods. When visiting a variable declaration node, it adds an entry which specifies the type of that variable] in a symbol table for that variable. Also when visiting a class declaration, it invokes TreeCompiler, which compiles the class into Java byte code. However, this compiling process alters the class and the generated byte code does not match the original source code of the class. The Evaluation Visitor is a tree visitor that evaluates each node of a syntax tree. This visitor is the one that performs the evaluation and execution of the program. It usually starts by invoking the main method of the compiled class. As it is evaluating and executing the program, it produces the intermediate code (MCode).

3. DEVELOPMENT OF CELIOTM

This section describes how CeliotM was developed by redesigning Celiot. It covers the following subsections: (i) How was Celiot redesigned to support memory addressing functionalities? (ii) How was Celiot modified to visualize memory address? (iii) How was Celiot modified to support pointer? (iv) How was Celiot redesigned to support dynamic memory allocation? (v) How was Celiot modified to support data structure features? (vi) How was Celiot modified to support data dynamic declaration? (vii) How was a C++compiler integrated in CeliotM framework to support direct compilation of data structure programs?

Redesigning of Celiot to support memory addressing functionalities

Celiot design focused on visualizing basic programming features based on the MTL by Mselle [19]. Celiot didn't include memory addressing and data structure functionalities when it was developed [13]. Thus, in order to enable Celiot to support the visualization of the data structure elements, the initial design of Celiot was modified to support memory addressing, pointers, and data structure declarations.Before redesigning Celiot to support data structures elements the first stage was to redesign Celiot to support memory addressing functionalities. The function of memory addressing Memory address is to find a permanent location of memory used by the program when you run it on your computer. This feature is a component of every pointer operation that occurs in any C++ program. There is no way to use pointer if you don't have it. Figure 5 shows the functional Structure of Addressing Engine.

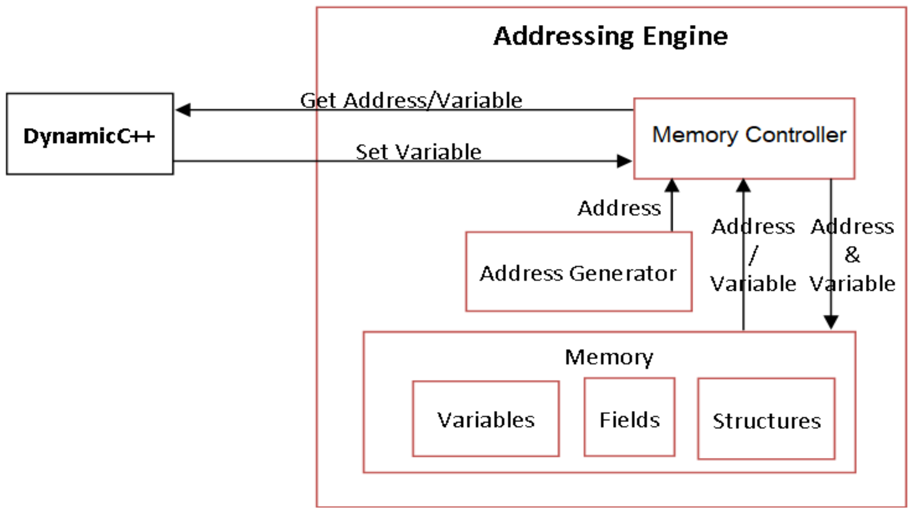


Figure 5. Functional Structure of Addressing Engine

The Addressing Engine consists of three components: Memory Controller, Address Generator and Memory. The Addressing Engine performs three main functions. First, it interacts with CeliotM interpreter (DynamicC++) to receive calls to either add a new symbol-address entry in the symbol table, or retrieve an address for a given symbol and vice versa. Secondly, by using an Address Generator it manages the addresses by generating a set of unique integers which resemble computer RAM addresses for each entry to be added. Lastly, through a Memory Controller it maintains a symbol table which contains the memory map.

A Memory Controller is the most active part of the Engine. It acts as a relay between the DynamicC++ and the Memory. It receives calls from DynamicC++ and serves them. Such calls are of two types, setting and getting. The setting call is used for adding a symbol-address entry in the memory. The getting call is used to retrieve either a symbol or an address associated with a symbol from the memory. Figure 6 shows the general design of CeliotM with the Addressing Engine. Therefore, the general execution flow of the tool is as follows:

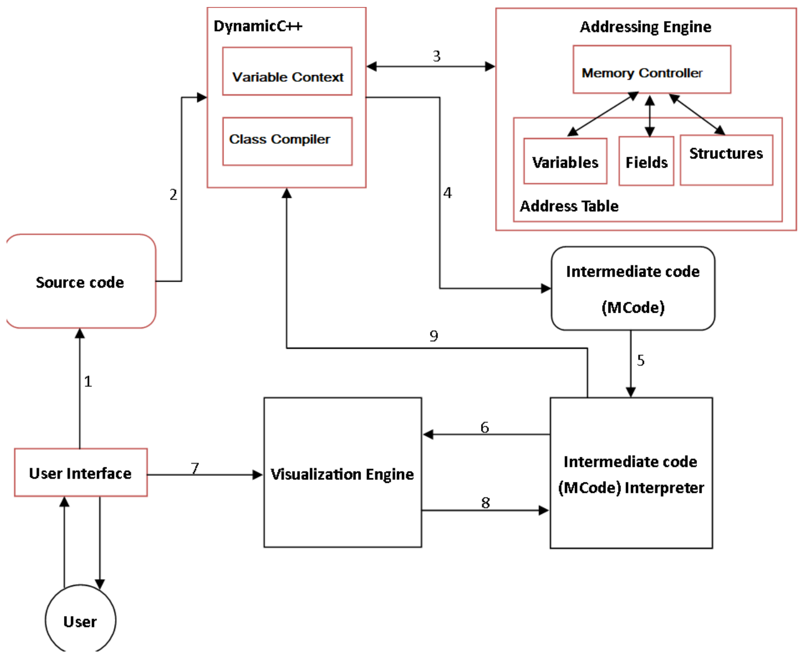


Figure 6. General Design of CeliotM with Addressing Engine

The tool accepts user programs entered through its interface (1) which may contain pointers and/or data structures features, and passes the source code of the program to the DynamicC++ for interpretation (2). During interpretation, two components which are responsible for initializing variables and structures in the virtualized memory, Variable Context and Class Compiler, send calls to the Addressing Engine (3) to generate and maintain an address for each variable and initialized structure encountered in the program. When the address needs to be resolved in other parts of the DynamicC++, respective calls are also sent to the Addressing Engine (3). Then, an intermediate code called MCode is generated and sent to its interpreter (4 and 5) ready for visualization. The MCode Interpreter directs the Visualization Engine (6) and then the program is visualized accordingly. Like in Celiot, a user is allowed to control the animation by playing, pausing, rewinding, or playing step-by-step the animation (7). When the program requires data input, a user can provide input to the program through the tool's interface (7, 8, and 9).

Facilitating Animation of Memory Addressing

The original design of Celiot visualizes the code by displaying the name of the variable, giving status as free reserved, or occupied based on MTL format. Celiot does not indicate variable memory addresses during animation. Therefore, it cannot visualize pointers and other objects that use pointers, such as linked lists and queues. In order to facilitate memory address animation in CeliotM , a new language construct called Address, and its actor called Address Actor were added. The two added items were used to visualize memory addresses. Additionally, variables and field actors were modified to accommodate Address Actors in them. Figure 7 (a) and (b) show how the variable and field actors were respectively modified. During the animations, some users prefer to see variable addresses, while others prefer to see the visualization of variables without displaying their actual addresses. Thus, in addressing such demand, two types of addressing modes were implemented. In the first mode, the addresses of variables are shown for clarification, and in another mode, they are hidden for simplification.

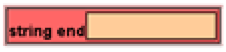


Figure 7 (a). Old Actor: Without address

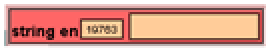


Figure 7 (b). Modified Actor: With address

Implementation of Pointer Operations in CeliotM

A pointer serves as a fundamental component of various data structure objects like linked lists, queues, and stacks. Due to this necessity, pointers had to be incorporated into CeliotM before the implementation of the actual data structures themselves. To implement pointer functionalities, the implementation of features supporting pointer-handling were required. Such features are: Pointer Declaration, Address of Expression (for retrieving the addresses of variables); Dereference Expression (for retrieving the value of a variable pointed by a given pointer); Pointer to Objects; and Pointer Member Access Expression (for accessing members of data structures pointed by a pointer).

For such implementation to take place, it was also necessary to enable new design to accept the source codes with pointer syntax and produces the MCode that contains the semantic of the codes. Since these features use syntax which must be developed, a series of modifications had to be performed in the Celiot interpreter and DynamicC++. Furthermore, some changes were made in the visualization engine to animate the pointer declarations and assignment operations such as address assignment, dereferencing pointing to objects and pointer member access.

In order to support declaration of pointers, new syntax rules had to be introduced in the variable declaration syntax so that the Parser could accept variable declarations with an asterisk sign (*) between the type and variable name as a syntactically correct statement. Now consider the following statements: (i) Type Identifier; (ii) type * Identifier. The first statement is a normal variable declaration. It has two fundamental tokens which are Type and Identifier. Type refers to the type of variable which is being declared, which can be int, float, string etc., and Identifier refers to the name of the variable. For pointer declaration, another token, asterisk sign, had to be added. That is in the second statement which is a pointer declaration with a single asterisk sign in between Type and Identifier. After a successful syntax analysis of such statements by the Parser, a syntax tree node called VariableDeclaration is created and added to the syntax tree. The node contains two other nodes which are Type and QualifiedName nodes. A QualifiedName node represents a valid C++ identifier and a Type node represents a variable or return type. A Type node can be of IntType, DoubleType, and ReferenceType etc.

In order to support Pointer types, another form of type called PointerType was introduced. PointerType contains a degree, which tells the degree of a pointer and represents asterisk(s) in between type and identifier in the syntax, and the targetType, which is the type of a variable that this pointer will be pointing to. This targetType can be IntType, DoubleType, or even another PointerType. Therefore, the VariableDeclaration node for a pointer declaration will have its type as a PointerType. Figure 8 shows a variable declaration with PointerType. The upward arrow in Figure 8 implies that modifications made to support pointer declaration still also support normal declaration.

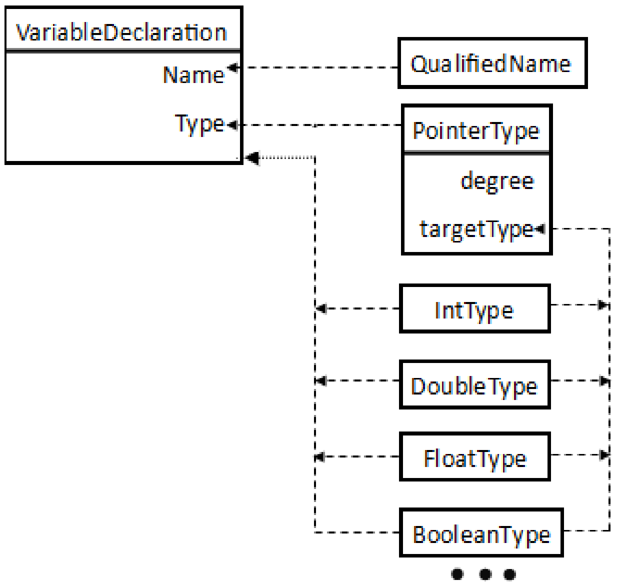


Figure 8. Variable Declaration with PointerType

Passing Pointer Declaration to Animation Engine

After a pointer variable has been checked for typing rules and added to a symbol table, the evaluation visitor produces MCode for pointer declaration which is sent to the animation engine. The same variable declaration MCode has been used to send a pointer declaration by just adding stars before a type field to indicate the degree of a pointer. If no stars are added, it means that the variable is not a pointer. If any star(s) are found, it means that the variable is a pointer of degree equal to a number of the stars. Thus, in case a pointer variable is declared as, e.g. int * p, and not a normal variable, the original Mcode format for Normal Variable Declaration (i.e. int var) in the form: 26|var|Initializer|0|int|false|Location) need to be modified to be: 26|pointer|Initializer|0|*int|false| Location.

Assuming that a user has compiled his program with a pointer declaration statement as follows: - int * p;. This program will be compiled as follows:- The Parser will detect the presence of an asterisk token after ‘int’ token and it will create a PointerType node with degree one and integer target type. Combining the type and an identifier after it, the Parser will create a VariableDeclaration node and attach it to a program’s syntax tree. The TypeChecker will add a variable ‘p’ to a symbol table stating that its type is integer (meaning that it will store addresses which are integers) and it is a first degree pointer pointing to integer variables. The Evaluation Visitor will produce MCode for the declaration in which it will add a single start before the variable’s type which is ‘int’, and the resulting type field will be ‘*int’. In order to accomplish the animation of pointer programs, a new Actor called PointerVariableActor was introduced. The actor extends VariableActor from Celiot 2 and adds a ReferenceActor in it so that it can point to its target. Figure 9 shows the Passing of Pointer Declaration to Animation Engine.

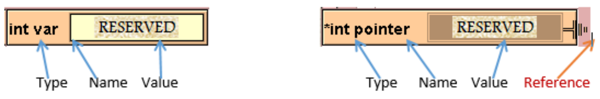


Figure 9. Passing Pointer Declaration to Animation Engine

Implementation of Address of Expression in CeliotM

Consider a statement p=&var; where p is a pointer variable, & is an address of operator and var is a normal variable. This is an expression which evaluates an address of a variable associated with the expression. The expression is primarily used to assign an address of a variable to a pointer, but can also be used in mathematical operations to calculate addresses of other memory locations. As described earlier, the Parser in the DynamicC++ in the version of Celiot did not support this type of assignment operation. Thus, in order to make this operation possible, it was required to introduce new grammar in the Parser to support this Expression. Now, from the statement p=&var; when the parser analyzes this statement, it will produce a single node known as SimpleAssignExpression. This node contains two other nodes which are right and left expressions.

The right expression here, as seen in the aforementioned statement, will be our AddressOfExpression. The expression is associated with a variable ‘var’ meaning that it has to be evaluated to an integer value which is an address of a variable ‘var’. It follows that in order to interpret this statement the Evaluation Visitor will evaluate the expression to an address of the variable ‘var’ (Refer to Getting a Symbol/Address section in Addressing to see what happens during this evaluation). After that, the Evaluation Visitor will write MCode to represent the expression and send it to the evaluation visitor. Here, a new MCode statement had to be introduced for this purpose. The MCode structure MCode for AddressOf Expression was as follows: 152|ExpressionCounter|StructureName|VariableName|Address|Type|Location

Note that the StructureName stands for the name of encapsulating structure, which is null if a variable does not belong to any structure/class; and Type stands for the type of this expression, which is always integer. In animation, the expression causes the copying of an address of a variable to where the expression assigns it, the Left Expression. Note that an address is copied ready to be transferred *to* where it is assigned. If the expression assigns it to a pointer, the value of that pointer will change, and its reference actor will point to a variable in the AddressOfExpression, as seen the second row in Figure 10.

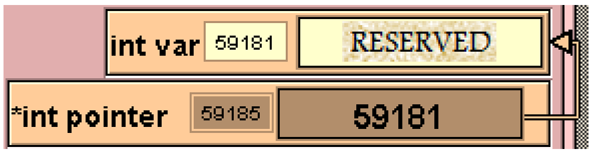


Figure 10. Implementation of Address of Expression in CeliotM

Implementation of Dereference Expression in CeliotM

When working with pointer, the dereference expression cannot only be used to retrieve a value of a pointed variable, but also to assign a value to the variable. That is to say that, statement '*p = 5' will assign a value of 5 to a variable pointed by pointer p since *p will evaluate the variable, hence making the expression equivalent to 'variable = 5'. Again, in order to include this feature in CeliotM, a new grammar was introduced in the Parser to support this Expression since it was not originally present in the previous Parser. The grammar was defined in such a way that the expression could be applied in all its cases as indicated previously. A new MCode statement has to be introduced. The new MCode is for a newly implemented feature of MCode for Dereference Expression as follows:

154|PointerDS|PointerName|VariableDS|VariableName|Location

Note that PointerDS stands for the name of the structure which contains the pointer; and VariableDS stands for the name of the structure which contains the variable. In animation, dereference expression is animated as blinking of the reference actor and a variable pointer by the reference actor. Figure 11 shows how the animation of dereference expression looks like.

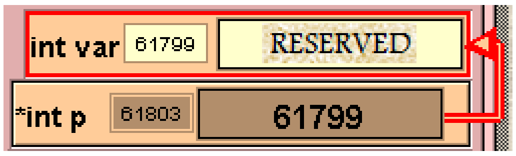


Figure 11. Animation of Dereference Expression in CeliotM

Implementation of Dynamic Declaration in CeliotM

Dynamic declaration is an important aspect of data structures programming. Since in data structures, a program uses lots of memory locations to store lots of data (like lists of numbers), it is in most cases hard to exactly predict the number of variables that the program will need at a specific time during execution, especially when the size and type of data are affected by user inputs. Furthermore, it is a good practice to create programs which declare variables when they are needed and delete them when they are not. Dynamic declarations give programs the power to do so, by allowing them to declare memory locations and delete them during execution. This feature was not an included feature in Jeliot 3 in an exposed way as required by C++ programming; just as Java does not expose machine resources to programmers as much as C++ does. All memory management activities are handled by the Java Virtual Machine. Celiot which was for C++ does not support the feature at all; since it neither supports Java (which has built in dynamic declarations) nor implements the feature for C++ language.

C++ programming supports three types of dynamic declarations which are primitive, structure/class and array dynamic declarations. Primitive Dynamic Declarations, which are used to declare single memory locations for simple data type. All dynamic variables of type int, float, double, etc. are associated with this type of the Structure/Class Dynamic Declarations, which are used to instantiate/initialize objects of structures and classes. These are used when an object is meant to be referred to by pointer (s). User-defined types using keywords class or struct use these declarations. Array Dynamic Declarations are used to instantiate dynamic arrays. Arrays always contain a number of cells of certain type. Their cells will also be dynamic and will be declared with appropriate types of declarations, depending on their type (primitive, structure/class or even another array).

All types of dynamic declarations use the same syntax, which is an assignment statement with pointer as a left expression and new expression as a right expression; the differences are in the data types of elements being declared, i.e. Pointer=new type; whereby ‘pointer’ is a pointer variable, ‘new’ the key word for a dynamic declaration, and ‘type’ for data type. Celiot had not implemented this feature; its predecessor, Jeliot 3, however, had implemented a new expression as the only way to initialize/allocate classes and uninitialized arrays. In Jeliot 3, primitive types are not allowed in the new expression except as part of array initialization. Once Jeliot 3’s Parser detects the expression, it creates one of the three nodes, SimpleAllocation, ClassAllocation or ArrayAllocation. SimpleAllocation node is created when a new expression is associated with initialization of a class which has been defined and given a class name in the program; ClassAllocation node is created when the associated class is anonymous, which is a class defined in its initialization expression; and ArrayAllocation node is created when the new expression is associated with an array.

After examining how the dynamic declaration works in Jeliot 3, the new expression features found in Jeliot 3 were modified to support C++. In order to make that possible, the primitive dynamic declaration was introduced by letting the Parser accept new expressions with primitive types. The new expression was also changed to make it accept class and structure declarations without parenthesis (which is a must in Jeliot 3, whereby a Java class must be initialized by calling at least its default constructor). Array Dynamic Declaration had minor changes to be made. Essentially, the primitive Dynamic Declaration feature was implemented in CeliotM by modifying the syntax of the new expression so as to accept primitive types like *int, float,*and *double*.

Implementation of Primitive Dynamic Declaration in CeliotM

Essentially, the primitive Dynamic Declaration feature was implemented in CeliotM by modifying the syntax of the new expression so as to accept primitive types like int, float, double, etc. With the general syntax pointed previously, the Parser was made to accept primitive types in the type field as a valid syntax. Assuming this as an example, pointer = new int; when the Parser encounters such a statement, it will create a SimpleAssignExpression as explained before, with the pointer variable (QualifiedName) as its left expression and the new expression as its right expression; and insert it in the syntax tree. The new expression will be represented by a node called PrimitiveAllocation, which was created for the primitive dynamic declarations. The node contains a type of variable to be created during execution which for the previously given example is type int.

When the node is evaluated in the TypeChecker, the TypeChecker will add an unnamed variable to a symbol table stating its type and address only. The address of that variable will then be returned and assigned to the pointer. From then on, the variable will be accessible for assignment and data access through the pointer. After that, the Evaluation Visitor will write MCode statement to the Animation Engine to animate the declaration and assign the address accordingly. Again, another MCode statement was introduced for this purpose.

156|Name|Address|Counter|Value|Type|isConstant|Location

MCode for Primitive Allocation

In the MCode, a Name field contains the address of the dynamic variable to make it possible for the Animation Engine to differentiate between static and dynamic variables. In Animation, when a variable's name is an integer, that is a dynamic variable whose name is not shown; the variable will therefore be accessed through a pointer only. Figure 12 shows the Primitive Dynamic Declaration.

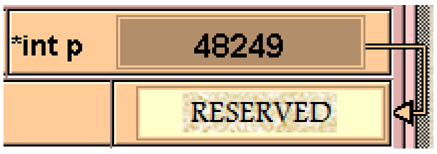


Figure 12. Primitive Dynamic Declaration

Implementation of Structure/Class Dynamic Declarations

The implementation of this type of dynamic declarations came after implementing Data structures. As explained previously, Jeliot 3 has a new expression feature which is used for instantiating Java classes. Since Java classes are not so different from C++ structures and classes (especially for introductory programming), the same features were modified and used for the declarations. In Jeliot 3, the new expression for class instantiation (assuming the default constructor) is expected to be as follows: var=new ClassName where ClassName is the name of a class being instantiated, which is hereby treated as the type of variable var; this kind of Type is referred to as Reference Type. While, in CeliotM the new expression for dynamic declaration of structures and classes was expected to be like: var=new StuctureName, where: StructureName is the name of a structure/class being declared dynamically, which is hereby treated as the target type of pointer var., which is also referred to as Reference Type. As it is seen in the syntaxes of the new expressions for Jeliot 3 and CeliotM above, there is a slight difference which is the necessity of placing parentheses after a class name. The Parser was therefore modified to accept the C++ syntax above as a valid syntax instead of that of Java.

Implementation of Dynamic Declarations of Arrays

Another feature that is included is the ability of the new CeliotM to compile and visualize Array Dynamic Declarations. The syntax of array dynamic declaration in C++ is similar to that one used to instantiate array objects in Java. Therefore, no modification of syntax was required for the arrays. During interpretation by the Evaluation Visitor, the same approach that was used in creating structure/class objects was used to create array. The only difference is that, for Arrays, Java has a specific API to create its objects. So, instead of creating equivalent class and call, its constructor, an instance is directly retrieved by calling a static class of a Java class, Array, while specifying the type and dimensions of the array. Likewise, after the object has been created, its ID is used as an address to add the object into the addressing table through the Addressing Engine. Then, the address is returned to be assigned to the pointer. Moreover, the Evaluation Visitor writes MCode statement to the Animation Engine to animate the object creation of the array and assign the address accordingly. The MCode has also been modified to add the same two fields, Address and is Dynamic, for the same purpose as indicated previously. The animation of dynamic declaration of arrays is similar to that of structures and classes. The address is shown and assigned to the pointer variable in assignment expression. Then, the pointer's reference will be made to point to the array object.

Implementation of Data Structures Declaratives in CeliotM

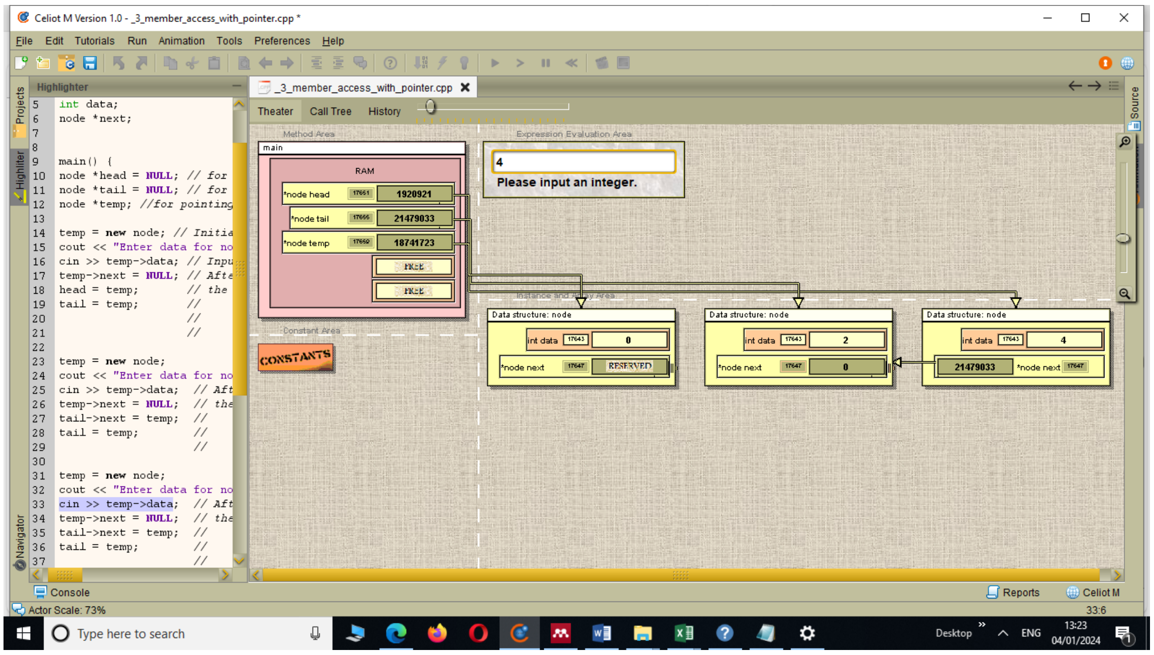
Data structures allow a user to define and use custom and compound data types in one's program. A data structure is a user-defined structured data type which can contain other data of mixed data types. This feature is not present in Jeliot 3. Jeliot 3 has Java classes instead, which are syntactically different from C++ structure, but functionally very similar to them. Although Celiot 2 was derived from Jeliot 3, it does not have data structures at all. This is because, Java classes are not working and C++ structures have not been implemented in it. C++ data structures are similar to Java classes except that, the default access level for data structure members is public, while in Java it is private and stated as 'package level accesses. Moreover, the syntax for data structures differs from that of Java classes. Table 1 shows the difference between a Java class and C++ structure.

Table 2

Java Class vs C++ Structure

`Class Data {`
`int intMember; // private to package`
`Public float floatMember; // public`
`}`
	`struct Data {`
	`int intMember; // public`
	`private: float floatMember; //`
	`public`
	`};`

As demonstrated above, the syntactic differences between C++ and Java are the keyword class and struct, a semicolon in structure after curly brackets (where declarations of variables/fields of the structure type might be placed) and member access specification. Since Jeliot 3 supports Java classes which are semantically not so different from C++ classes and structures, the Celiot interpreter had to be modified to accept data structure syntax and then converted to a java class for interpretation. In order to enable the Parser in CeliotM to support a syntactically correct C++ structure definition, a ClassDeclaration node had to be created for the structure. The node carries all the required information about a class. The same node was used by Jeliot 3 to capture class blueprint. This node was modified so that it could carry definitions of C++ classes and structures instead of those of Java classes. However, advanced features of C++ classes were not implemented. Furthermore, a flag was added in the node in order to state whether it was representing a class or a data structure. Unlike other PV tools, CeliotM animates data structure program source codes using MTL form. In this format, at any instant, the actor shows the status of program execution in computer memory using three memory (RAM) cells: free, reserved, and occupied. Figure 13 shows the snapshot of visualization of a linked list program in CeliotM.



```
int data;
node *next;


main() {
node *head = NULL; // for
node *tail = NULL; // for
node *temp; //for pointing

temp = new node; // Initia
cout << "Enter data for no
cin >> temp->data; // Inpu
temp->next = NULL; // Afte
head = temp;       // the
tail = temp;       //
                   //
                   //

temp = new node;
cout << "Enter data for no
cin >> temp->data;  // Aft
temp->next = NULL;  // the
tail->next = temp;  //
tail = temp;        //
                    //

temp = new node;
cout << "Enter data for no
cin >> temp->data;  // Aft
temp->next = NULL;  // the
tail->next = temp;  //
tail = temp;        //
                    //
```

4

Please input an integer.

Figure 13. A snapshot of a visualization of a linked list program in CeliotM

Integrating a C++ Compiler in CeliotM

CeliotM does not visualize all data structure concepts. In order to enhance learners' engagement and interest in using CeliotM, it was decided to combine CeliotM with the C++ mainstream compiler.In order to reduce the ECL imposed on users due to the split attention effect resulting from the switching overheads, during requirement analysis phase, users suggested the integration of CeliotM with the C++ mainstream compiler. To accomplish this task, a stand-alone standard C++ compiler application was first found; followed by developing an interface through which CeliotM could communicate with it to send a novice’s program for compilation and receive responses from it. Since most of the expected users of CeliotM were using Windows OS, the compiler that was going to be integrated had to be compatible with Windows environment.

Novices could be instructed to use tools like MinGW or Cygwin to download and install the compiler (on windows) and a way to locate the compiler from CeliotM could be provided. However, the tools were somehow tricky to handle. They could confuse a novice and make him opt for other IDEs. At the end of this little research about the compiler to be integrated, it was decided that a free small-sized command line compiler which ships with Borland C++ IDE be used. The compiler was extracted from IDE by installing IDE (in a chosen Windows PC) with minimal installation options. Only command-line tools, include-files, some libraries and other important components were chosen. This resulted into a folder structure of only 18.8 Mbs which was then compressed to a size of 3.92Mbs. After the compiler was found, an interface to connect the compiler with CeliotM was designed and implemented. The main roles of the interface were to: start the command line tool of a runtime pc; connect its streams with CeliotM’s I/O and error console; execute the compiler to compile novice’s programs; and execute the created executable programs. Figure 14 shows the process of integrating a C++ Compiler within a CeliotM PV tool.

Therefore, the C++ Compiler was integrated into CeliotM by extracting it from the C++ IDE. Thus, unlike other PVs, in which the mainstream compilers and visualization software are two separate programs[12][27]. In CeliotM, you can directly run your program on your machine from the Tool without needing any other IDE to do so. When a C++ compiler is used, all responses from the compiler including error messages will be displayed in both the error panel and I/O-console of CeliotM. The major benefit of integrating compiler in CeliotM was to improve the efficiency of using CeliotM in teaching and learning to program. This is because by combining both PV and compiler in one package, users will avoid unnecessary switching between CeliotM and a C++ compiler.

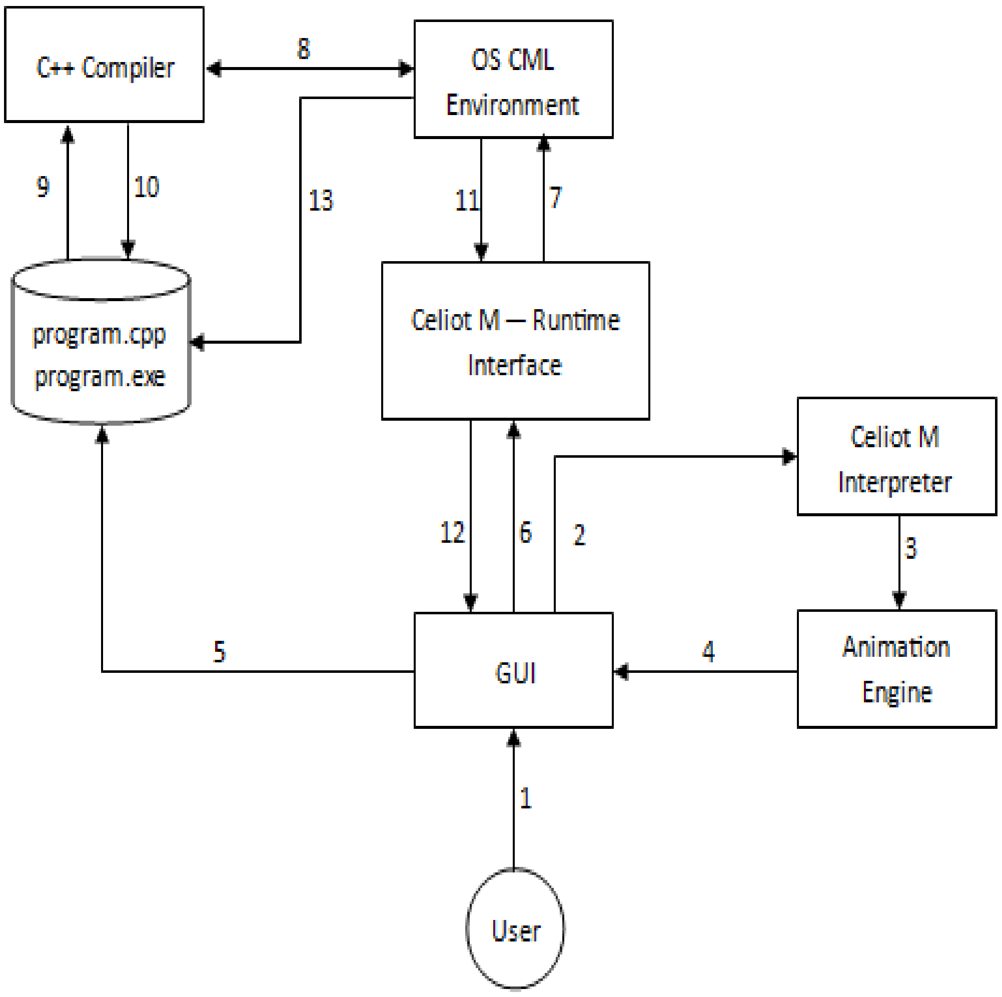


Figure 14. Integration of C++ Compiler in CeliotM

After a novice has finished editing the program, he/she may choose either to animate the program, or to compile it and create its executable version (1). All these options are provided on the GUI of CeliotM. When he/she chooses to animate his/her program, things go as normal. The program source code is handed to the Interpreter (2) which creates intermediate code, MCode, which is then handed to the Animation Engine (3) for displaying animations (4). Nevertheless, when he/she chooses to compile his/her program, things go differently now. CeliotM firstly saves the file being edited into the disk (5), and then communicates with its runtime interface telling it the name and path of the file to be compiled (6). The interface, upon receiving the message, starts the command line environment of the runtime OS, Windows, and connects the Input/ Output (IO) Console and Error Panel of CeliotM to IO Streams and Error Stream of the command line respectively. Then, it executes the compile-command on the running command line environment telling it the path and name of the compiler to-be-created as executable file and source code file (7) as shown in Figure 15.

[compiler-executable] **–n**[*output-executable-file*] [*source-file*]

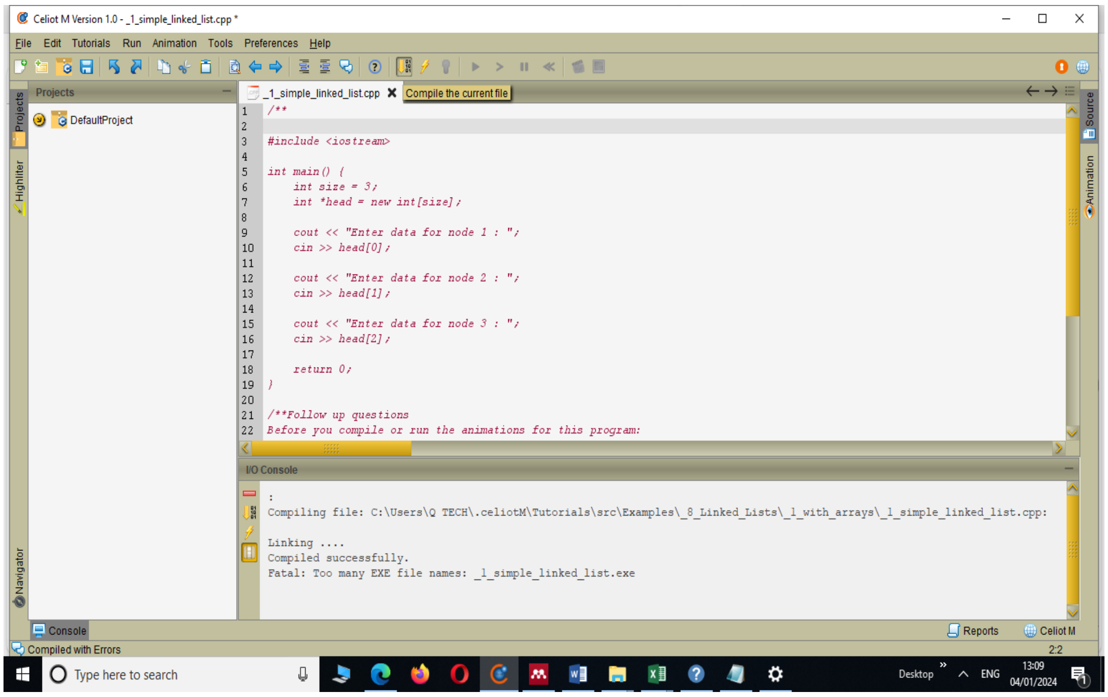
Where *[compiler-executable]* – **path and name of the installed compiler, similar to** *[user-home-directory]\.celiotM\ BC5\BIN\bcc32.exe*

[output-executable-file] – **path and name of the output executable file, similar to** *[project-folder]\build\[source-name].exe*

[source-file] – **path and name of the source file, similar to** *[project-folder]\src\[source-name].cpp*

Figure 15. Execution of the Compile-command

The compiler is then started and arguments to compile the program are passed (8). The compiler reads the source code from the disc (9), by using a source path provided in its parameters and compiles it. If the compilation is successful, the compiler writes an application file of the program into the provided path (10), and returns a successful message to the command line (8). The command line propagates the message to CeliotM's Input Output (I/O) console (11) and (12) and the message is printed. Again, if an error occurs, an error message is propagated to the error panel. Likewise, to run the created application, upon clicking a run button, the interface starts the command line environment and connects the streams with the GUI. Then, it executes a command which tells the command line the path of an executable file corresponding to a given source file. The command line then executes the program. Thus, with the compiler integrated within CeliotM, a novice does not need to use another tool/IDE to learn programming. Instead, the user will only create /edit all the source code programs in CeliotM editor and then decide whether to run animations by using CeliotM or use the inbuilt C++ compiler to compile and run the program. Figure 16 shows a snapshot of an array program that has been compiled successfully in CeliotM.



```
/**

#include <iostream>

int main() {
    int size = 3;
    int *head = new int[size];

    cout << "Enter data for node 1 : ";
    cin >> head[0];

    cout << "Enter data for node 2 : ";
    cin >> head[1];

    cout << "Enter data for node 3 : ";
    cin >> head[2];

    return 0;
}

/**Follow up questions
Before you compile or run the animations for this program:
```

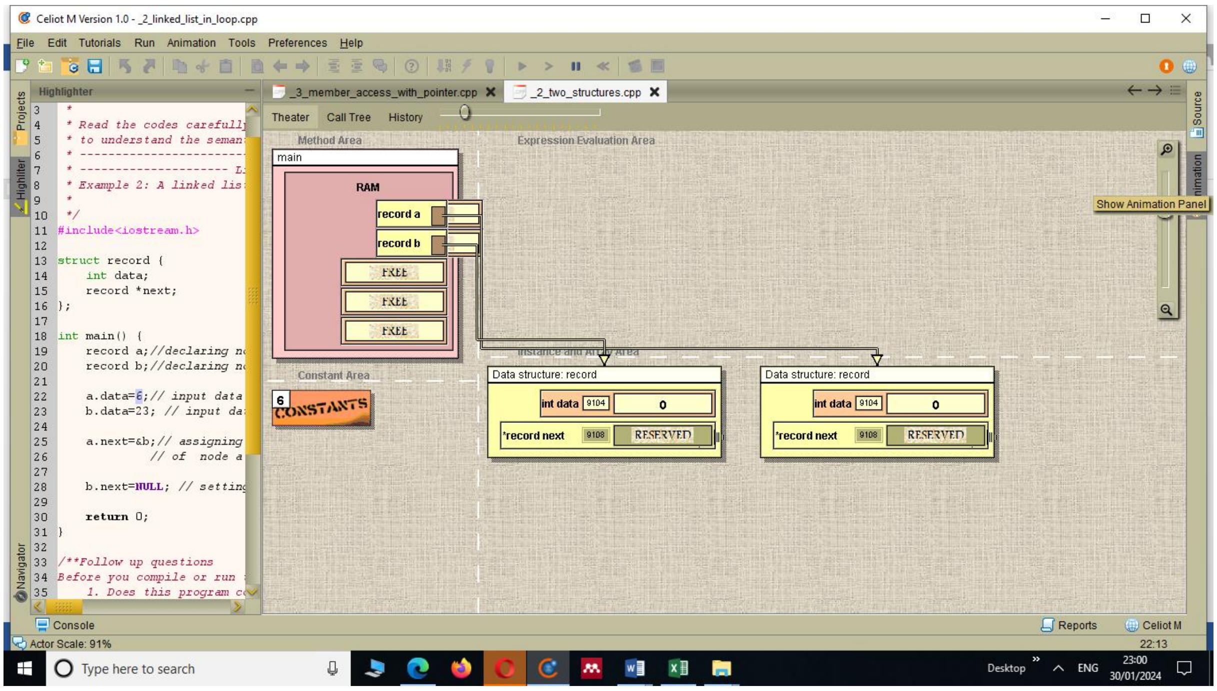
```
:
Compiling file: C:\Users\Q TECH\.celiotM\Tutorials\src\Examples\_8_Linked_Lists\_1_with_arrays\_1_simple_linked_list.cpp:

Linking ....
Compiled successfully.
Fatal: Too many EXE file names: _1_simple_linked_list.exe
```

Figure 16. A snapshot of compilation of program in CeliotM

Implementation of the Zooming Feature in CeliotM

Controlling the pace and visibility of animation is an essential step towards reducing ECL. Animation components in Celiot were in a fixed-large size. Since Celiot was intended to show minimal animations of basic programming concepts, there was no need to change the size. However, in data structures animations, object creation and referencing concepts demand more components; hence more space is required. To resolve this, a zooming feature was introduced by which components could be made smaller to accommodate more of them in the same space. This feature was implemented by modifying animation actors of the Animation Panel, Theatre, which were previously drawn with statistically defined dimensions and locations. They were modified to allow scaling of the dimensions and their relocation. The whole scaling process was performed after the actor had received a new scaling factor from user's input. Thus, for actors who contain other actors, also called actor containers, a parent actor/container propagates the received scaling factor to its child. Although the transformation Application Programming Interface (API) of Java Software Development Kit (SDK) could be used to zoom the whole graphics object of the Animation Panel at once, and life could have been easier, that would not yield as perfect results as the used approach did. This is because the transformation of the graphics by scaling distorts the clearness of lines and texts of the animations. Scaling is performed by multiplying a new scaling factor with the original dimension of an actor. The scaling factor is then saved to be used later when another scaling factor is received. Since display dimensions were in integral values which would drop all decimal points resulting from scaling calculations, the scaled decimal value was also saved together with the scaling factor for accuracy of scaling later on.

When another scaling factor is received from a user, it is compared to the previous scaling factor to determine if the actor should scale or not. Scaling is only performed when a new scaling factor is different from the previous. After an actor has been scaled, its location is then recalculated to position it at an appropriate location. Likewise, if an actor is a container, it tells its child to recalculate their locations. In order to zoom a given animation in CeliotM, a user pulls a slider found on the animation panel. A scaling factor is calculated from a position of the slider and sent to actors of the animation panel. Calculations of a scaling factor are performed in such a way that the largest scaling factor will allow actors to zoom to slightly larger sizes than the original, and the smallest scaling factor will allow actors to zoom to as smaller sizes as the texts and lines can be seen clearly. It was expected that improving the ability to view the animation would reduce much visual efforts among the users, and thus increase the motivation and interest to use CeliotM in teaching and learning DSA. Figure 17 shows an example of a linked list program with two nodes being visualized with a zoom-in feature turned on in CeliotM.



```
  *
  * Read the codes carefull
  * to understand the seman
  * -----------------------
  * -------------------- L
  * Example 2: A linked lis
  *
  */
#include<iostream.h>

struct record {
    int data;
    record *next;
};

int main() {
    record a;//declaring n
    record b;//declaring n

    a.data=6;// input data
    b.data=23; // input da

    a.next=&b;// assigning
              // of  node a

    b.next=NULL; // setting

    return 0;
}

/**Follow up questions
Before you compile or run
    1. Does this program c
```

Figure 17. Zoom Feture in CeliotM

Implementation of User-defined Explanations in CeliotM

Another feature which was included in CeliotM is user-defined explanation. This feature was added to help learners to map visualisations with algorithm being executed. By using this feature, a user could be provided with ability to add a comment which explains a certain line in the source code. The initial plan was to implement the feature by making the animation engine generate explanation for every statement. However, this plan was cancelled since it was actually a more sophisticated approach than one could think. This is because. if it was not done carefully and effectively, it could end up confusing a novice even more, because the explanation would mostly be based on explaining about the animation actors So, lastly, it was concluded that the best design approach of this feature, for the sake of a novice, was to let instructor or any user writes the explanation about the codes which would be shown as the program was being visualized.

In order to enable this service, a new feature, which is called CommentList was added. This feature facilitated interaction with CeliotM GUI, Editor and Animation Engine. The component holds a list of explanation statements for source code lines as defined by a user. Figure 18 shows a CommentList as it interacts with GUI, Editor and Animation Engine.

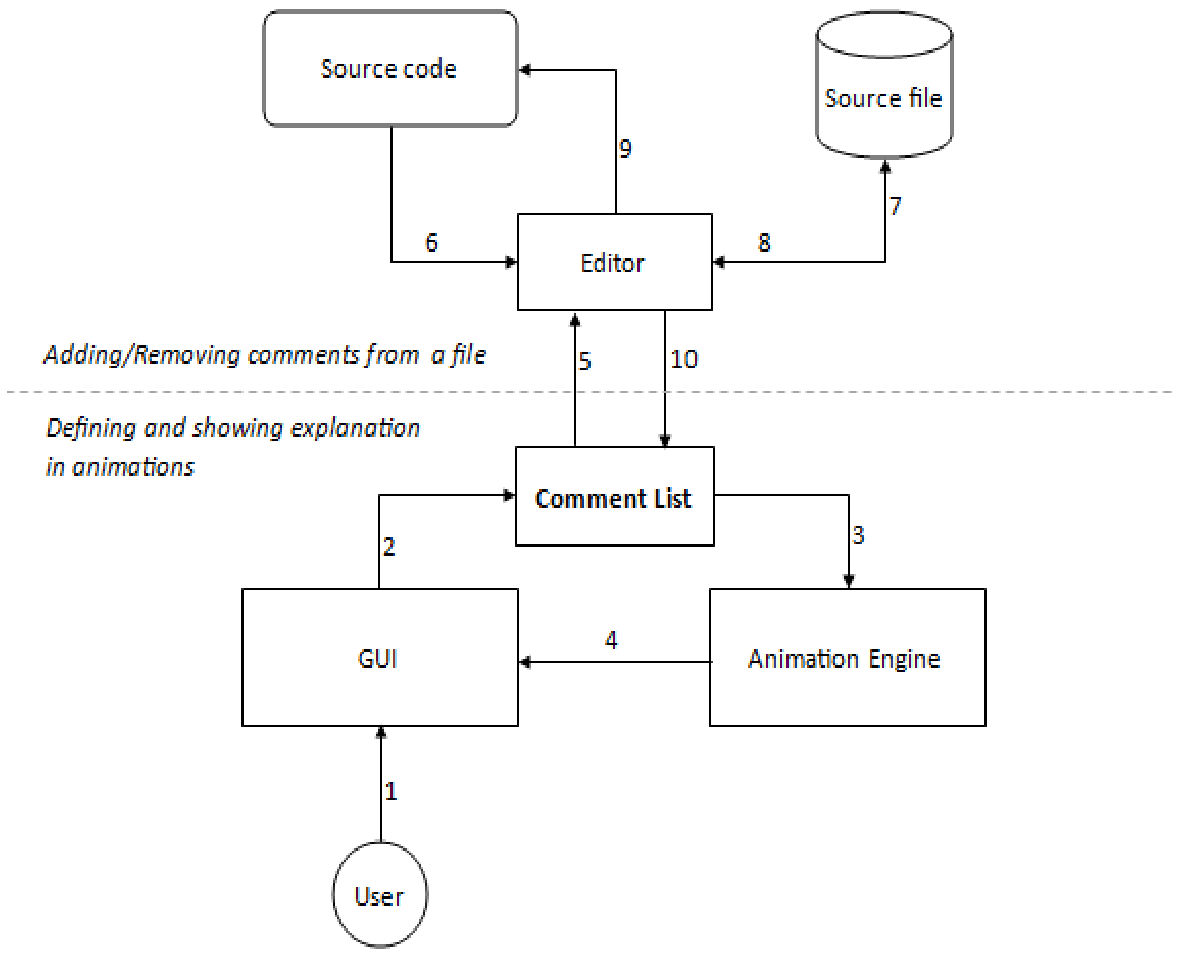
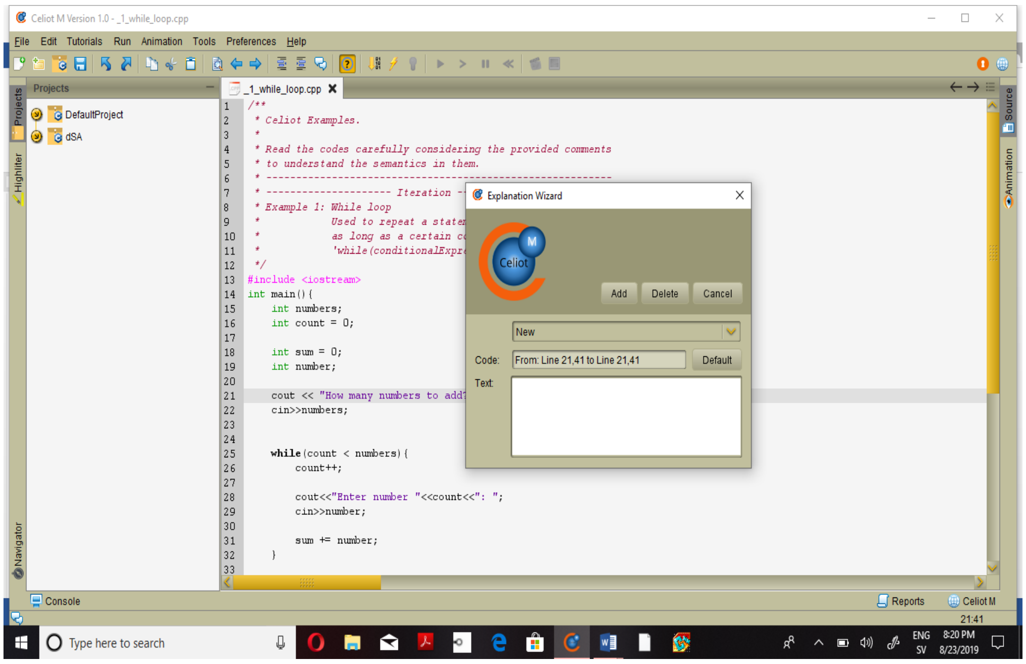

Source code
Source file
9
7
6
Editor
8
Adding/Removing comments from a file
5
10
Defining and showing explanation in animations
Comment List
2
3
GUI
4
Animation Engine
1
User

Figure 18. Comment List as it Interacts with GUI, Editor and Animation Engine

Figure 18 depicts a design approach of the feature which allows a user to input his/her narrative through the GUI (1). He/She places a cursor in a line of a statement that he wants to explain and clicks on explanation button (?) on a toolbar. The dialog box will open requiring him to enter explanations and add them. Upon adding, the explanation text and its corresponding source code location are added to a comment list (2). Up to there, the explanations are ready to be used during animation. When the program is being animated, the animation engine reads the comment list for every statement that it is animating (3). When the reader comments result into an explanation text, the text is shown to a user through an actor called *MessageActor* (4). How, then, are these explanations shipped to another user, a student? When a user saves his file by clicking on a save button on a toolbar, the Editor retrieves the explanations, if any, from the CommentList (5) and converts it to C++ comments. It then adds the comments at the end of a source code text (6); and lastly saves the text in a corresponding source file (7). When the file is opened again (8); say by a student, the Editor separates explanation comments from the source text, showing the source code without the appended comments (9), while adding the explanations into the CommentList (10). The explanations will then be shown during animation as explained earlier. Figure 19 shows a typical explanation wizard for adding explanations alongside the code.



```
/**
 * Celiot Examples.
 *
 * Read the codes carefully considering the provided comments
 * to understand the semantics in them.
 * ---------------------------------------------------------
 * -------------------- Iteration -
 * Example 1: While loop
 *            Used to repeat a state
 *            as long as a certain co
 *            'while(conditionalExpre
 */
#include <iostream>
int main(){
    int numbers;
    int count = 0;

    int sum = 0;
    int number;

    cout << "How many numbers to add?
    cin>>numbers;


    while(count < numbers){
        count++;

        cout<<"Enter number "<<count<<": ";
        cin>>number;

        sum += number;
    }
```

New
Code: From: Line 21,41 to Line 21,41
Text:

Figure 19. Including User and Celiot Explanations

Inclusion of Tutorial Module in CeliotM

The tutorial module consisted of examples, exercises (problem-solving) and error tracing exercises. All these categories were grouped together in a set called Tutorials. These tutorials were made by creating a folder with the same structure as other CeliotM project folders and adding source files in it. The folder was then compressed to a zip file and added in a CeliotM file structure ready to ship to users. Tutorial's source folder contains three folders for the three categories of tutorials, namely; examples, exercises, and error tracing exercises. Examples folder contains nine folders for nine C++ topics. The folder and file names are preceded by indexing numbers so as to arrange them in an advancing order rather than alphabetical order. Likewise, the Exercises folder contains nine folders for nine C++ topics. Finally, the Error Tracing Exercises folder contains six folders for six C++ topics. In the user's machine, CeliotM decompresses the Tutorials folder, and extracts its contents to the folder with the same name in the user's home directory. It then adds a menu in its GUI's menu bar titled 'Tutorials' from which a user will be able to access all tutorial contents.

CeliotM menu contains three submenus for the three categories of tutorials which further contain other submenus for all folders and, lastly, menu items for all files related to their respective file structure. In order for the user to open a tutorial file, he/she clicks on a menu, Tutorial, and then navigates to his/her desired file. The user will then clicks a menu item of the desired file and waits for the file to be opened in CeliotM's source code editor. CeliotM receives the click event and pulls a file path related to the clicked menu item. It then sends the path to the editor for the opening of the file. The tutorial module provided by CeliotM also includes introductory programming examples and exercises. Celiot examples are unique in that such exercises have been written in a more natural way and are guided by both instructive directions that insist the learner to read the questions carefully and examine clearly what the program does before.

The CeliotM User Interface

Figure 20 shows the user interface of CeliotM. As shown in Figure 20, the theatre panel of CeliotM consists of four parts labeled as follows: (1) Method Area, (2) Expression Evaluation Area, (3) Constant Area, and (4) Instance and Array Area.

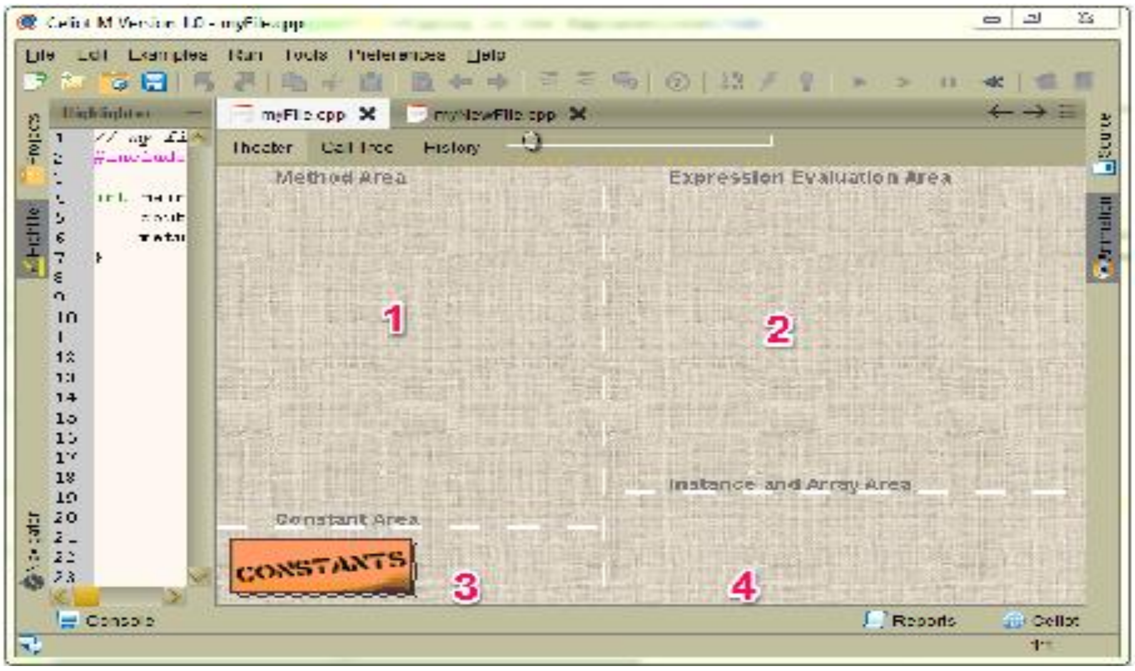


Figure 20. The Theatre with its Four Parts

The CeliotM interface parts are separated by white-dashed lines appearing as margins between them. Each part has a heading on top of it, describing what type of actors it is displaying. The function of the Method Area is to display functions called by your program. The function of the Expression Evaluation Area is to show all actors animating the program expressions and their evaluations. These expressions are Arithmetic expressions, Logic expressions, Function calls, etc. The expressions receive values from the Method Area and other parts of the theatre, evaluate them and send the results to appropriate destinations. The constant area contains two actors, the Constant Box, and a Global Context. The constant box is used to generate the constant values that the user used in his/her programs. Finally, in Instance and Array Area, it enables a user to see Actors of arrays and data structures instances. It follows that when you want to animate your program after you have finished editing it, you will click on the 'Animation' button on the right of the extended toolbar. Then, the animation panel will open, telling you that CeliotM is compiling your program. After the compilation is finished, you will click on a play button on the main toolbar to start the animation. There it comes to the theatre on which the animation components call Actors and drawn.

System testing and evaluation

In order to ensure system's quality, functionality, usability, performance, and security, the resulting CeliotM prototype was tested in the laboratory and refined for any improvements before being deployed for use in actual class settings for empirical evaluations. The system was tested through various testing using appropriate tools and met the required standards. To ensure the correctness of individual system components, unit testing was performed. The system was divided into modules, and each separable module was subjected to alpha testing where by testing performed by professionals or developers and beta testing which tested by friends and normal user. The effectiveness of using CeliotM in teaching data structures has been empirically evaluated in practical settings. Six class experiments were performed to validate if the use of the CeliotM enhance learning DSA among novices. Results show that the use of CeliotM framework improved student's comprehension and motivation in learning data structures concepts [17] [18][19].

4. COMPARISON OF CELIOTM WITH OTHER PV/AV TOOLS

Table 2 compares the features implemented in CeliotM with other PV/AV tools reviewed in this study. The plus (+) sign means presence of a given feature while negative (-) sign means the absence of feature.

Table 3

Comparison of Implemented Features CeliotM with other PV/AV tools

Feature	Jeliot 3	Celiot	PITON	Courseware	CeliotM
String	-	+		+	+
Dynamic declarations	-	-	-	+	+
Data Structures (struct)	-	-	-	+	+
Structure/Object Instantiation	Java objects class.	-	-	+	C++class and struct
Pointer Member Access (->)	-	-	-	+	+
Attractive GUI	-	-	+	-	+
Memory status base on MTL	-	+	-	-	+
Congruent visual structure for DSA	-	-	-	-	+
Integration with standard compiler	-	-	+	-	+
Zoom capabilities	-	-	-	-	+
System and user defined explanation	-	-	Implement color during visualization	-	+
Enhanced tutorial	-	-	+	-	+

As shown in Table 2, CeliotM shares several features with other PV/AV tools. However, it differs from other tools. CeliotM supports data structure components in both class and struct keywords in C++. It supports visualization of the stack, queue, and linked list based on the C++ imperative approach in MTL format, by depicting the status of memory, whether it is free or not, before visualizing each line of the code. This feature differentiates CeliotM from other PV and AV tools. So far, no AV/PV can visualize data structures constructs based on MTL. In CeliotM, there is full functionality of pointer, following the inclusion of the memory address engine as a new feature. These features are neither supported in Jeliot 3 nor in Celiot [13] together with all its families.

Unlike other PV/AV tools such as PITON and Courseware, in CeliotM, users can magnify the visualized object's view using a zooming option. CeliotM also supports new form of dynamic explanations for visualizations. Although Jeliot 3 with explanation [31] also offers some explanations, the explanations in CeliotM are unique in such a way that they allow flexibility as they can be used for both provisions of dynamic comments and output prediction. To help learner avoid ECL resulting from using visualization tools and actual compiler separately, CeliotM is integrated with standard C++ compiler. The major benefit of integrating compiler in CeliotM was to improve the efficiency of using CeliotM in teaching and learning to program. This is because by combining both PV and compiler in one package, users will avoid using a separate IDE tool. The decision to include the C++ compiler is in line with requirements from other researchers such as[12] and [32]. Thus, with the C++ compiler integrated into the tool (CeliotM), a novice does not need to use another tool/IDE to learn to program. This arrangement also makes CeliotM capable of actively engaging learners as a PV, AV, and IDE. CeliotM supports both imperative and object oriented programming (OOP) programs with the both struct and class keywords. However, Jeliot 3 supports the creation of Java Objects with keyword class only, while Celiot supports neither class nor structure. In CeliotM, there is full functionality of pointer, following the inclusion of the memory address engine as a new feature.

Furthermore, unlike in other PV/AVs, the dynamic visual images in CeliotM correspond with those used in MTL learner-driven instructional materials. Thus helps reduce misconception in using visualization. Although other PV/AVs such as PITON, and jGRASP [33] supports compiler compiler like CeliotM, jGRASP does not visualize a program based on the MTL format [19], and it works more as an IDE than an education-supporting tool intended for novices. In addition, differnt from other PV/AV tools, in CeliotM, users can magnify the visualized object's view using a zooming option.CeliotM also supports new form of dynamic explanations for visualisations. Although Jeliot 3 with explanation [31] also offers some explanations, the explanations in CeliotM are unique in such a way that they allow flexibility as they can be used for both provisions of dynamic comments and output prediction.

To help learner avoid ECL resulting from using visualization tools and another IDE separately, CeliotM is integrated with standard C++ compiler. The major benefit of integrating compiler in CeliotM was to improve the efficiency of using CeliotM in teaching and learning to program. The decision to include the C++ compiler is in line with requirements from other researchers such as [34] and [32]. Thus, with the C++ compiler integrated into the tool (CeliotM), a user does not need to use another tool/IDE to learn to program. This arrangement also makes CeliotM capable of actively engaging learners as a PV, AV, and an IDE tool.

Another main enhancement in CeliotM design was the inclusion of tutorial modules. In CeliotM, a set of exercises and examples have been embedded to assist a self-learning user. Although both Jeliot 3 and Celiot included examples, they were not included in a complete and simple-to-use way as in CeliotM. The examples provided in Jeliot 3 [31] and Celiot [13] were just a randomly selected set of programs to show a user some of the capabilities of the tools. However, CeliotM provides a well-organized set of C++ programs that explain to a novice learner the different concepts of C++ programming from the introductory level to data structures and algorithms. When compared to other PV tools, CeliotM examples and exercises are unique in that they have been written more naturally. They are guided by a set of guidelines that guide learners in doing exercises. The aim of including this guideline was to increase learner’s attention and concentration.

5. CONCLUSIONS AND PROSPECTS FOR FURTHER RESEARCH

This paper describes how CeliotM was developed and implemented to support visualization of DSA concepts. CeliotM was developed by re-designing the structure of Celiot using reuse-oriented software engineering development method. CeliotM can now visualize and compile linear data structures in C++ such as array, records, queue, stacks, and linked lists. It also supports dynamic and primitive data structure declaration. CeliotM can visualize programming constructs by using both variable name and address. CeliotM integrates PV, AV and IDE in one working environment. It can therefore be used as a standalone compiler, animation tool or both. Evaluation of the CeliotM has shown that it greatly helped students to understand data structures courses. The development the CeliotM is expected to enhance the motivation and interest of instructors and students to use PV tools in teaching and learning data structures. This study has discussed how CeliotM was modified to support linear data structures elements, memory addressing, pointer and dynamic declarations. The practical contribution of this work is to provide a new PV tool that support teaching and learning basic programming and data structures concepts in C++. The design implication of this improvement in CeliotM brings several prospects in the development of programming learning support tools.

CeliotM platform currently visualizes and compiles data structure concepts in C++ programming language only based on MTL format. One challenge that still prevents the wide adoption of visualization too is that exist is the limited number of tools that can work as IDE and Visualizers yet support more than one programing in one working environment, leading to few adoption of PV tool in learning programming. The availability of such environment help engaging more support to more novices programmer including such those who studies C++, Python, C, and Java. Future studies should consider redesigning CeliotM to support visualization and compilation of data structures concepts in other programming languages such as Python, Java, and C.

Currently, CeliotM support text animation only. To enhance learner's engagement in using program visualization tool. CeliotM may be integrated with audio explanation in future. The tool also currently support linear data stuctures. Future work should consider extending the capabilities of CeliotM to visualize advanced data structure elements such as trees and graphs based on MTL diagrams.

REFERENCES (TRANSLATED AND TRANSLITERATED)

[1] S. Dehnadi and R. Bornat, "The camel has two humps (working title)," *Middlesex Univ. UK*, pp. 1–21, 2006.

[2] L. Layman, Y. Song, and C. Guinn, "Toward Predicting Success and Failure in CS2: A Mixed-Method Analysis," *ACMSE 2020 - Proc. 2020 ACM Southeast Conf.*, no. 1, pp. 218–225, 2020, doi: 10.1145/3374135.3385277.

[3] E. Fouh, M. Akbar, C. A. Shaffer, and V. Tech, "The Role of Visualization in Computer Science Education," 2012.

[4] B. Park and D. T. Ahmed, "Abstracting Learning Methods for Stack and Queue Data Structures in Video Games," *2017 Int. Conf. Comput. Sci. Comput. Intell.*, pp. 1051–1054, 2017, doi: 10.1109/CSCI.2017.183.

[5] J. Danielsiek, Holger, Paul, Wolfgang, Vahrenhod, "Detecting and Understanding Students ' Misconceptions," pp. 21–26, 2012.

[6] T. L. Naps *et al.*, "Exploring the Role of Visualization and Engagement in Computer Science Education Report of the Working Group on &quot; Improving the Educational Impact of Algorithm Visualization &quot;," *Acm*, vol. 35, no. 2, pp. 131–152, 2002.

[7] M. Mladenovi and M. Agli, "The impact of using program visualization techniques on learning basic programming concepts at the K – 12 level," no. July, 2020, doi: 10.1002/cae.22315.

[8] C. D. Hundhausen, S. A. Douglas, and J. T. Stasko, "A meta-study of algorithm visualization effectiveness," *J. Vis. Lang. Comput.*, vol. 13, no. 3, pp. 259–290, 2002, doi: 10.1006/jvlc.2002.0237.

[9] K. Romanowska, G. Singh, M. A. A. Dewan, and F. Lin, "Towards Developing an Effective Algorithm Visualization Tool for Online Learning," *2018 IEEE SmartWorld, Ubiquitous Intell. Comput. Adv. Trust. Comput. Scalable Comput. Commun. Cloud Big Data Comput. Internet People Smart City Innov.*, no. March 2020, pp. 2011–2016, 2018, doi: 10.1109/SmartWorld.2018.00336.

[10] E. Vagianou, "Program Working Storage : A Beginner ' s Model," pp. 69–76, 2006.

[11] P. Bellstrom and C. Thoren, "Learning how to program through visualization: A pilot study on the bubble sort algorithm," in *2009 Second International Conference on the Applications of Digital Information and Web Technologies,* 2009, pp. 90–94.

[12] E. Isohanni, *Visualizations in Learning Programming Building a Theory of Student Engagement*, no. November. 2013.

[13] M. Masoud, "Combining Memory Transfer Language (Mtl) and Jeliot 3 To Evolve a Visual Language for Teaching and Learning Programming," PhD, Thesis. The University of Dodoma, 2015.

[14] P. Dewan, "Discovery-based Praxes : Channelling the User- Interface of an Industrial-Strength

Programming Environment to Formally Teach Programming," pp. 341–342, 2017.
[15] A. Moreno and N. Myller, "Producing an educationally effective and usable tool for learning, the case of the jeliot family," 2003.
[16] W. I. Osman and M. M. Elmusharaf, "Effectiveness of Combining Algorithm and Program Animation : A Case Study with Data Structure Course," vol. 11, pp. 155–168, 2014.
[17] E. Elvina, O. Karnalim, M. Ayub, and M. C. Wijanto, "Combining program visualization with programming workspace to assist students for completing programming laboratory task," *J. Technol. Sci. Educ.*, vol. 8, no. 4, pp. 268–280, 2018, doi: 10.3926/jotse.420.
[18] M. S. Joy, "Jeliot 3 in a Demanding Educational Setting," vol. 178, pp. 51–59, 2007, doi: 10.1016/j.entcs.2007.01.033.
[19] L. Mselle, "Formalization of memory transfer language with C, C++ and java on the mold of register transfer language," *ICISA 2014 - 2014 5th Int. Conf. Inf. Sci. Appl.*, pp. 0–3, 2014, doi: 10.1109/ICISA.2014.6847404.
[20] J. Sorva, V. Karavirta, and L. Malmi, "A Review of Generic Program Visualization Systems for Introductory," vol. 13, no. 4, 2013.
[21] J. Sorva, V. Karavirta, and L. Malmi, "A review of generic program visualization systems for introductory programming education," *ACM Trans. Comput. Educ.*, vol. 13, no. 4, 2013, doi: 10.1145/2490822.
[22] R. A. Nathasya, O. Karnalim, and M. Ayub, "Integrating program and algorithm visualisation for learning data structure implementation Integrating program and algorithm visualisation for learning data structure implementation," *Egypt. Informatics J.*, no. November, 2019, doi: 10.1016/j.eij.2019.05.001.
[23] I. Sommerville, *Ninth Edition*. Pearson, 2011.
[24] I. Sommerville, *Ninth Edition*, Ninth Edit. Boston: Addison-Wesley, 2011.
[25] T. Naps *et al.*, "Evaluating the Educational Impact of Visualization," 2003.
[26] N. Myller, "The Fundamental Design Issues of Jeliot 3," 2004.
[27] A. Moreno, N. Myller, and E. Sutinen, "Visualizing Programs with Jeliot 3," 2004.
[28] M. M. Mtaho Adam., Mselle Leornard., "Effectiveness of Using a Congruent Visualization Framework on Learning a Data Structures Course," *Educ. Pedagog. J.*, vol. 6, no. 2, pp. 60–72, 2023.
[29] A. B. Mtaho, "Effects of using problem-solving guide and explanatory support in program visualization tool on reducing students' misconceptions in learning data structure concepts," *Bull. Soc. Informatics Theory Appl.*, vol. 7, no. 2, pp. 125–140, 2023.
[30] A. B. Mtaho, "The Impact of Combining Follow-Up Questions and Worked Examples In Program Visualization Tool on Improving Students ' Held Mental Models of Pointers ' Value and Address assignment," 2022, doi: 10.23951/2782-2575-2022-2-53-64.
[31] A. Moreno, *Animation Re-designing Program Animation*, no. 149. University of Eastern Finland, 2014.
[32] N. J. Coull, "SNOOPIE : Development of a Learning Support Tool for Novice Programmers within a Conceptual Framework," University of St Andrews, 2008.
[33] J. H. C. Ii, D. Hendrix, and D. A. Umphress, "jGRASP : An Integrated Development Environment with Visualizations for Teaching Java in CS1 , CS2 , and Beyond," pp. 3–4, 2004.
[34] E. Isohanni, "Visualizations in Learning Programming Building a Theory of Student Engagement," *Tampere Univ. Technol.*, 2013.

Text of the article was accepted by Editorial Team 14.01.2024

РОЗРОБЛЕННЯ НАВЧАЛЬНОГО ІНСТРУМЕНТУ ПРОГРАМУВАННЯ CELIOTM ДЛЯ НАВЧАННЯ КОНЦЕПЦІЙ СТРУКТУР ДАНИХ У С++ ПРОГРАМІСТІВ-ПОЧАТКІВЦІВ

Адам Б. Мтахо
кандидат комп'ютерних наук, викладач кафедри інформаційно-комунікаційних технологій
Арушський технічний коледж, м.Аруша, Танзанія
ORCID ID 0000-0002-6997-3332
abasigie@yahoo.com

Масуд М. Масуд
кандидат комп'ютерних наук, заступник директора Департаменту розвитку ІКТ та технологій
Танзанійська організація промислових досліджень та розвитку, м.Дар-ес-Салам, Танзанія,
bigutu@gmail.com

Леонард Дж. Мселле
кандидат комп'ютерних наук, доцент кафедри комп'ютерних наук та інженерії
Університет Додоми, м.Додома, Танзанія
mselel@yahoo.com

Анотація. Програмування є основною навичкою в навчанні комп'ютерних наук (CS). Воно також є корисним предметом в інженерних та природничо-наукових курсах. Однак викладання та вивчення комп'ютерного програмування не є простим завданням. Про це свідчить той факт, що більшість студентів стикаються з проблемами та труднощами в розумінні концепцій програмування та способів їх застосування в реальних життєвих ситуаціях. Ще гірша ситуація з курсом "Структури даних та алгоритми" (DSA) - курсом програмування просунутого рівня, який є обов'язковим для будь-якого студента, що вивчає CS. Цей предмет надто складний для розуміння новачками через його абстрактну та динамічну природу. Для подолання цих труднощів було створено кілька інструментів візуалізації алгоритмів (AB), які допомагають новачкам зрозуміти структуру даних. Однак педагогічна ефективність використання таких інструментів не була успішною, оскільки вони є менш привабливими для вивчення DSA. У цій статті описано, як було розроблено навчальний інструмент програмування CeliotM для полегшення вивчення концепцій структури даних у C++. Розробка CeliotM була досягнута завдяки використанню підходу програмної інженерії, орієнтованого на повторне використання. CeliotM було розроблено шляхом перепроєктування інструменту візуалізації програм (PV) Celiot - інструменту для навчання програмуванню, який підтримує навчання програмуванню на C++. У такий спосіб, використовуючи Java як мову, оригінальна версія Celiot була перероблена для підтримки компіляції та візуалізації різних елементів структури даних у форматі мови передачі даних (MTL); крім цього додано декілька функцій для залучення учнів, зокрема вбудований компілятор C++ та анімаційні пояснення. У результаті було створено навчальний програмний засіб CeliotM, який візуалізує та компілює об'єкти структур даних, такі як черги, стеки та зв'язані списки мовою програмування C++. Емпіричні результати оцінки використання CeliotM у навчанні структурам даних та алгоритмічним концепціям показують, що використання такого інструменту покращило розуміння студентами програмування та запропонувало більш привабливий досвід навчання для програмістів-початківців. Найбільший внесок цієї роботи полягає у створенні навчального інструменту для викладання структур даних у C++, який може працювати як компілятор, програма та інструмент візуалізації алгоритмів у тандемі. Вона також є цінним ресурсом для навчання програмуванню, пропонуючи ефективний і надихаючий підхід для початківців до розуміння фундаментальних концепцій програмування і структур даних.

Ключові слова: CeliotM; візуалізація структур даних; MTL; навчання програмуванню