UDK 378.14
DOI: 10.23951/2782-2575-2022-2-53-64

# 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

***Adam Basigie Mtaho***

*Arusha Technical College, Tanzania*

**Abstract.** Previous studies have shown that the lack of a useful mental model of pointers is one of the reasons why many novice programmers fail the data structures course. This study had two main objectives: to analyze the status of mental models of pointers (focusing on value and address assignment); and to evaluate the impact of combining worked-examples and follow-up questions in CeliotM program visualization (PV) tool in the learning of pointers. The subjects of the study were sixty-two second-year undergraduate students taking a course on data structures (PMT 221) at the College of Natural and Mathematical Sciences (CNMS) of the University of Dodoma. Data were collected using pretest and posttest questionnaires. The collected data were analyzed using descriptive statistics. The results showed that 56.5% of the students had incorrect mental models of pointers. The results also showed that using the proposed strategy improved the students' mental models of pointers from 56.5% to 87.1%. These results contribute to our understanding of the most common misconceptions that novice students may have when learning pointers. The findings of this study confirm previous studies that when the new innovative teaching strategies are used in combination with PV tools in teaching and learning programming can help improve students' programming comprehension.

**Keywords:** *programming, program visualization, threshold concept, pointers, mental model, follow-up questions*

## Introduction

Programming is the core competency in computer science (CS) education. To master programming, a beginner must learn and understand various concepts of programming threshold concepts. The Association for Computing Machinery (ACM) recognizes pointers as one of the threshold concepts that any student who learns to program must understand well [1]. Furthermore, pointers are prerequisite for learning data structures and other advanced programming topics in CS education [2, 4]. Being threshold means that they are concepts that act as bridge stones, i.e., once a student masters them, he/she can quickly understand the rest of the topics in the course.

On the other hand, a student who does not understand such concepts well may get stuck and be unable to learn a new concept [5]. Simply put, no student learning to program can implement data structures such as linked lists, stacks, queues, and trees if such student holds an incorrect mental model of pointers. Although pointers are "bridge stones" for learning data structures and other advanced topics in programming courses, they are still considered one of the most difficult concepts for novice programmers to learn and understand [3, 6].

Several studies have investigated students' misconceptions when learning pointers and suggested methods to address them [1, 3, 4, 7]. However, despite these efforts, few studies have investigated the viability of mental models of pointers in novice programmers using mental model theory [8]. To fill this research gap, this study used mental model theory [9] to investigate the viability of mental models of pointers in novice programmers (considering pointer value and address assignment); and evaluate the impact of combining follow-up questions and worked examples in PV tool on improving students' held mental models of pointers' value ad address assignment, hereafter referred to as the mental models of pointers.

## Literature review

### *Mental Model: Theoretical framework*

Students who learn basic programming concepts have problems in formulating a correct mental model of program execution [10, 11]. In programming learning, the term mental model is often used to describe how knowledge is organized in people's minds. Essentially, a mental model is a person's internal (mental) representation of real-world objects and systems [12]. It is the learner's understanding of the hidden information process underlying the code and operations in the computer.

Mental model studies are widely used in introductory programming. According to Ma [10], students constantly face the challenge of creating a new mental model when learning new concepts [10]. Studies have shown that most students, when learning to program, lack viable mental models when learning other threshold programming concepts. For example, in one study, McCauley et al. [13] found that students learning data structures tend to build an incorrect or inconsistent mental model of recursion and iteration when implementing a linked list.

Danielsiek et al. [14] also found that some students had the misconception that every binary tree is a search tree. In addition, some students were found to have a false mental model of assignment (i.e., they confused = in mathematics with that in programming. The term "false mental" model implies that the way a student thinks about code may differ from that of the notional machine [16]. It is well known that students with false mental models cannot understand programming [10]. To help such students build a viable mental model of the key programming concepts and gain programming competencies, previous studies have suggested the use of visualization tools [14, 22, 26].

### *Mental Model Theory*

The term mental model refers to the mental representations or beliefs of real things. It refers to how people understand the world and things around them, how they work, and how they are used and operated [17]. They are representations of reality that people use to understand certain phenomena. According to Johnson-laird [9], when the mind is confronted with a new learning task about a particular phenomenon or entity, the object or concept to be learned is represented by a token in the model. Depending on the phenomenon or concept under study, multiple models are usually created in the mind when learning a particular phenomenon. The properties of its tokens represent the properties of a particular entity, and the relationships between tokens represent the relationships between entities in people's minds (memory). When the learner needs to apply the model to solve a new problem, his or her cognitive system must create one or more models that represent states or situations consistent with the premises and "read" the inference from the model(s). If a conclusion is successfully read, a deception phase follows in which an attempt is made to construct a mental model that contradicts the proposed conclusion. If this is unsuccessful, the conclusion is accepted, and the mental model is updated [9].

It follows that when we learn a new concept or experience an event, the degree of accuracy of what we express to others depends on how correctly or incorrectly we have initially internalized the original knowledge and how we succeed in retrieving the "correct" mental models from several held models we have in our minds. However, the quality and accuracy of the mental models we hold are not always consistent and static. They are constantly changing and depend mainly on how they are stored and organized in our long-term memory. According to Norman [12], their modification and evaluation are inevitable to ensure that the mental models we possess remain useful for learning [12].

### *Mental models and learning*

Mental model theory helps us understand our learning environments in several ways [18]. According to Edwards-Leis [18], mental models help us explain, predict, control actions and thoughts, diagnose, communicate, and remember. The explanatory function enables understanding and strategy selection because mental models "facilitate cognitive and physical interactions with the environment, with others, and with artifacts." The predictive function enables problem-solving in new situations. The control function provides a platform from which decisions can be made. The diagnostic function of mental models enables students to develop metacognitive awareness and assimilate new knowledge with existing knowledge. Finally, the communication function allows others to see, understand, and remember the externalization of a person's mental models. How well a person accesses or retrieves the desired mental model, or part of it, depends on the efficiency of the memory process and the relevance of the perceived relations [18].

### *Visualization in learning programming*

According to Petre and de Quincey [24], the term visualization refers to the use of a graphical representation of information to assist human comprehension of and reasoning about that information. Wright & Laboratories [25] argue that visualizations are more useful in learning programming since they display information in a format that is closer to the user's mental representations of problems and will allow data to be processed in a format closer to the way objects are manipulated in the real world; they are easier to understand for novice programmers [25]. They are also useful for describing intricate programs that are difficult to describe with text languages, so graphical specifications may be more appropriate [26]. Through the use of visualizations, a learner can help a learner to understand the logic behind an idea by showing the intermediate steps and transitions [26].

Visualization in programming is grouped into two fundamental categories, namely, program visualization (PV ) and algorithm visualization (AV), depending on whether they animate a code or algorithm under execution by relating to the code or not. PVs are used for visualizing either the construction or execution (and sometimes both) of a program [27]. They are used to depict the source code or the state of a program or its execution with the visual means. Examples of PVs are Jeliot 3 [28] and Jype [29]. AVs are used for visualizing data structures and their operations according to a particular algorithm and transition between those states during the execution of the algorithm. Examples of AVs are AP-SA [30] and SDA-TRAKLA2 [31]. Research shows that using PV/AV tools help beginners to develop a viable understanding of the notional machine that runs the programs that they write, thereby helping them to learn how to program and improve both programming and problem-solving skills. [26, 32]. Due to the benefits that visualizations have brought to learning programming, several studies have examined their usage in learning programming. Closest to our works are those of Ma et al. [10], Adcock et al. [4], Rørnes [19], and Almadhoun and Parham-Mocello [20].

### *Related works*

Ma et al. [10] conducted a study to investigate the viability of novice programmers' mental models of basic programming concepts, focusing on value and reference assignments. The results showed that one-third of the students appeared to have "non-viable" mental models of value assignment, and only 17% held a viable mental model of reference assignment. To address this problem, Ma et al. [10] proposed a teaching technique that integrates PV and cognitive conflict strategies. A number of studies have found that using the cognitive conflict strategy is potentially effective in learning programming because it improves learner engagement and helps novice programmers develop a better understanding of key programming concepts.

Adcock et al. [4] investigated the impact of using a pair programming approach on undergraduate programming instruction. The approach combined instruction with hands-on exercises to allow beginning students to visualize these difficult concepts and understand how they work and why they need them. Results showed that using this approach reduced student frustration and improved retention in learning programming.

Kumar and Road [2] studied the effects of using software to learn pointers. The software helped students learn how to implement pointer programs. The effects of using the software teaching tool were evaluated using a pretest and a posttest. The results showed that the average performance of the posttest exceeded that of the pretest by 100%.

Rørnes [19] surveyed the different mental models associated with students' misconceptions in learning reference tasks and references in a Python programming course. The results showed that students had incorrect mental models related to reference assignments and/or calling a function. The results of this study were very useful because they helped instructors learn how students understand reference assignments and references when learning to program

Almadhoun and Parham-Mocello [20] conducted the study to investigate the status of conceptual and procedural mental models of eleven students who were engaged in creating programs with linked lists in the C programming language. The results showed that no participants held correct mental models of a simple linked list in C.

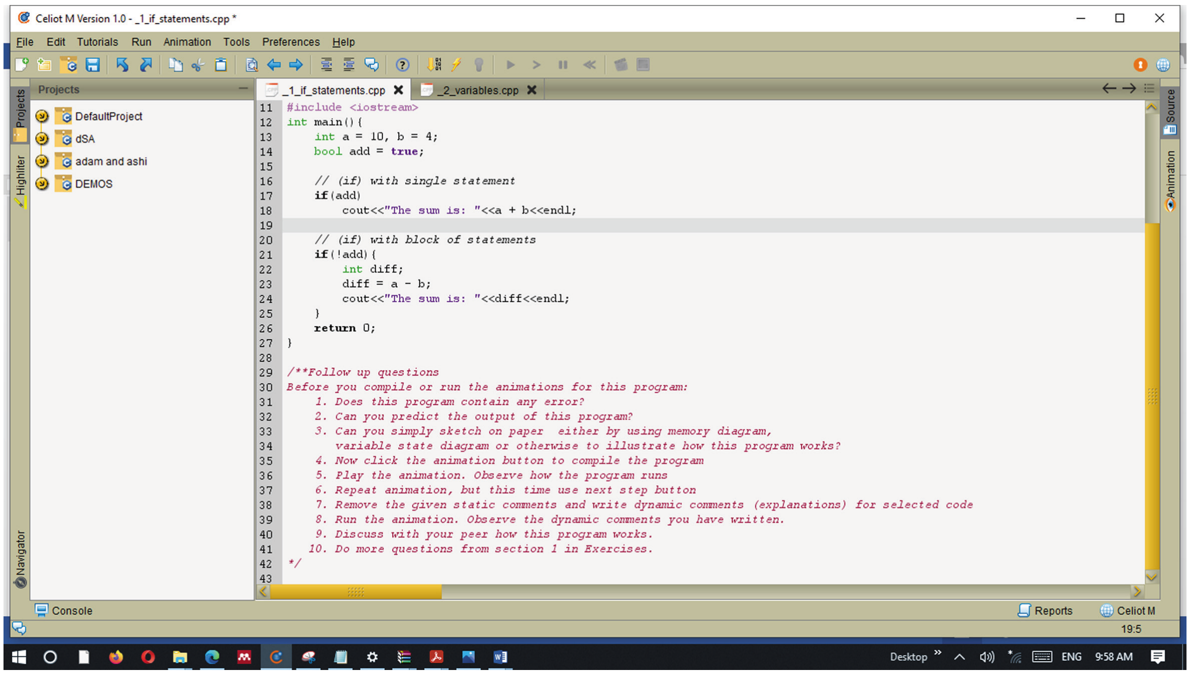
The reviewed studies have shown that the majority of students taking programming courses still hold incorrect mental models of threshold programming concepts. However, they have also shown that visualizations can help improve students' mental models of both basic and threshold programming concepts.

## *Overview of the proposed approach*

The proposed approach involves the use of follow-up questions embedded in worked examples in a CeliotM (PV) tool [21]. CeliotM is a PV tool intended for teaching and learning computer programming in C++ for novice programmers. CeliotM supports the visualization and compilation of both data structures and basic programming concepts in the C++ programming language. The tool offers a code view and an animation view. The source code view provides a text editor for writing code. The animation view provides the dynamic view of the program. CeliotM allows the user to work with different examples of threshold concepts in programming. Once a user opens an example, it appears in the source code view. Follow-up questions usually appear after each program example in the CeliotM. These follow-up questions were programmed into the tool to provide a natural interface to texts and other content in the PV tool, as recommended by Sorva et al. [22]. It follows that before a student compiles or runs the animations for any example program, he or she must answer the following questions:

1. Does this program contain any errors?
2. Can you predict the output of this program?
3. Can you simply sketch on paper, either with a memory diagram, a variable diagram, a state diagram, or some other way to illustrate how this program works?
4. Now click the animation button to compile the program
5. Play the animation. Watch how the program runs
6. Repeat the animation, but this time use the next step button.
7. Remove the specified static comments and write dynamic comments (explanations) for the selected code.
8. Run the animation. Observe the dynamic comments you wrote.
9. Discuss with your colleague how this program works
10. Work on other questions from this section.

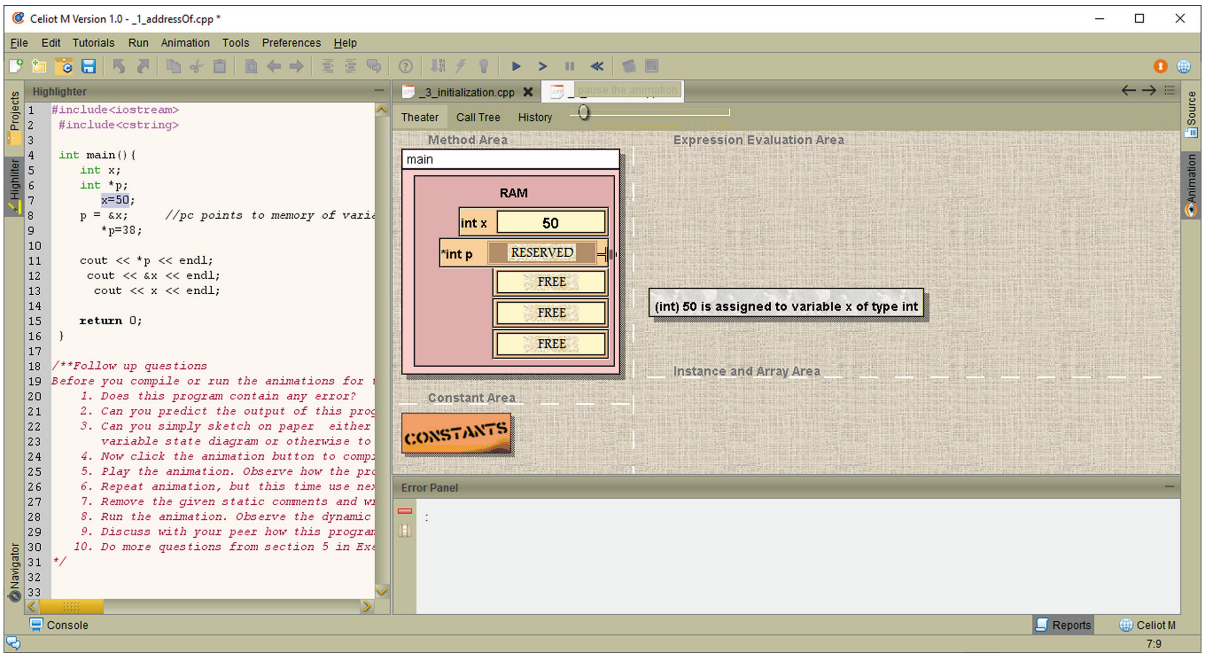
Figure 1 shows the view of the source code view and follow-up questions in CeliotM



```
#include <iostream>
int main(){
    int a = 10, b = 4;
    bool add = true;

    // (if) with single statement
    if(add)
        cout<<"The sum is: "<<a + b<<endl;

    // (if) with block of statements
    if(!add){
        int diff;
        diff = a - b;
        cout<<"The sum is: "<<diff<<endl;
    }
    return 0;
}

/**Follow up questions
Before you compile or run the animations for this program:
    1. Does this program contain any error?
    2. Can you predict the output of this program?
    3. Can you simply sketch on paper  either by using memory diagram,
       variable state diagram or otherwise to illustrate how this program works?
    4. Now click the animation button to compile the program
    5. Play the animation. Observe how the program runs
    6. Repeat animation, but this time use next step button
    7. Remove the given static comments and write dynamic comments (explanations) for selected code
    8. Run the animation. Observe the dynamic comments you have written.
    9. Discuss with your peer how this program works.
   10. Do more questions from section 1 in Exercises.
*/
```

Fig. 1. Source code, animation views, and follow-up questions in CeliotM

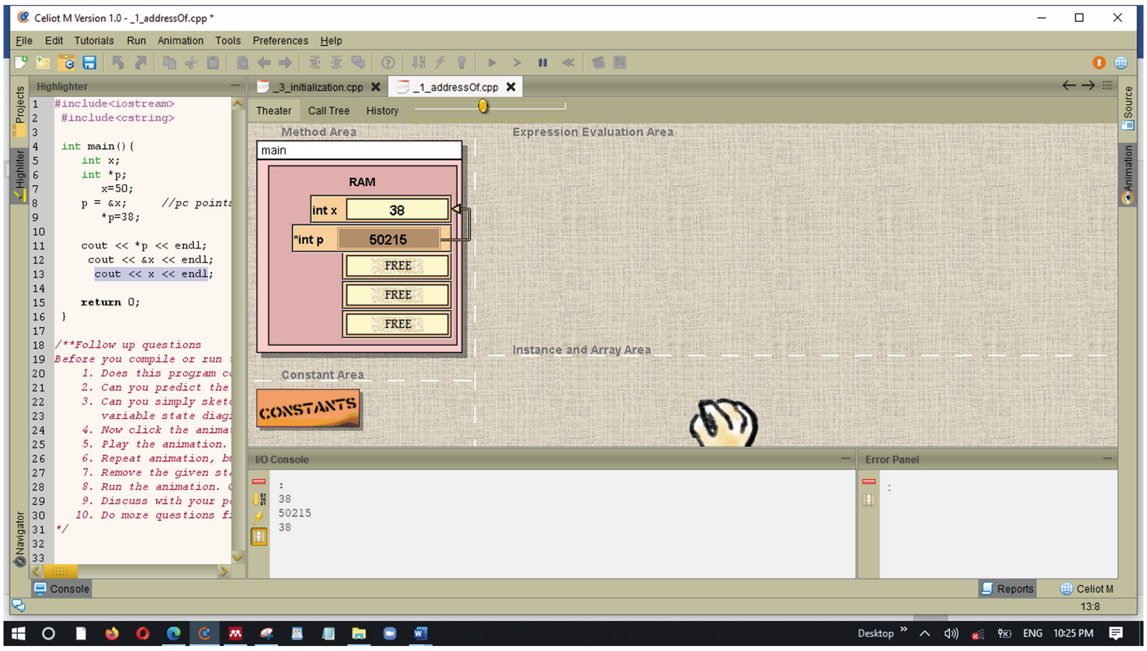
Once a student has answered the follow-up questions, he/she can switch to view the animation. Figure 2 shows a visualization of a pointer program in CeliotM. The animation in the method area displays the visualization of lines of code by indicating the status of the memory (RAM), whether it is free, reserved, or already assigned, as the program runs. For example, the program shown in the CeliotM source code view in Figure 2 has currently been executed from line 1 to line 7. Meanwhile, variable *x* is now assigned int 50, while pointer *p** has not been assigned to any value, so its memory location is reserved.



```
1  #include<iostream>
2   #include<cstring>
3
4   int main(){
5       int x;
6       int *p;
7          x=50;
8       p = &x;      //pc points to memory of varia
9          *p=38;
10
11      cout << *p << endl;
12       cout << &x << endl;
13        cout << x << endl;
14
15      return 0;
16  }
17
18 /**Follow up questions
19 Before you compile or run the animations for t
20     1. Does this program contain any error?
21     2. Can you predict the output of this prog
22     3. Can you simply sketch on paper  either
23        variable state diagram or otherwise to
24     4. Now click the animation button to compi
25     5. Play the animation. Observe how the pro
26     6. Repeat animation, but this time use nex
27     7. Remove the given static comments and wr
28     8. Run the animation. Observe the dynamic
29     9. Discuss with your peer how this progran
30    10. Do more questions from section 5 in Exe
31  */
32
33
```

Fig. 2. The visualization of a pointer program in CeliotM

Thus, through the use of animation, system-defined explanation, and other features, the learner understands the programming logic and improves programming comprehension. Figure 3 shows the final state of the animation of a given pointer program. It also displays the output of the program.



```
#include<iostream>
 #include<cstring>

 int main(){
    int x;
    int *p;
       x=50;
    p = &x;     //pc points
       *p=38;

    cout << *p << endl;
     cout << &x << endl;
      cout << x << endl;

    return 0;
 }

/**Follow up questions
Before you compile or run
    1. Does this program c
    2. Can you predict the
    3. Can you simply sketc
       variable state diag
    4. Now click the anima
    5. Play the animation.
    6. Repeat animation, bu
    7. Remove the given st
    8. Run the animation.
    9. Discuss with your p
   10. Do more questions fi
*/
```

```
:
38
50215
38
```

Fig. 3. The final animation state of a pointer program in CeliotM

## Materials and methods

A mixed research design was used in this study. The pretest and posttest questionnaires were used to examine the status of the mental pointer models that the students held before and after the intervention. The experimental method was used to examine the effect of the proposed approach on the improvement of students' mental pointer models.

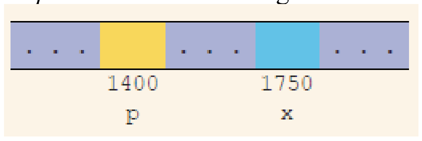
### *The structure of the pretest and posttest questionnaires*

The structure of the pretest and posttest questionnaires

The pretest questionnaire was used to examine the status of students' held mental models of pointers. It consisted of a multiple-choice question with a set of options for the mental models and an option to indicate the correct mental models if the participant did not find a correct option in the given ranges. The researcher's first work was to conduct an independent analysis of students' possible (a-priori) responses to the questions in the questionnaire that would correspond to a student's mental model type. These priorities are listed in Table1. Each student was assumed to have a latent cognitive structure underlying his or her responses to the questions in the questionnaire, referred to as "mental models." This method was adopted by Dehnadi and Bornat [23]. Before the pretest questionnaire was given to the participants, it was first validated. Then, a small pilot test was conducted in which students completed the questionnaire and expressed their opinions about the usefulness of the instruments. The comments given were processed by changing some options of the questionnaire. The posttest was constructed to measure the same cognitive level of mental models as the pretest. However, the format of the posttest questions and the level of cognitive ability measured remained the same as in the pretest. What was changed were the variable names and assigned values, but the programming logic remained the same. The maximum duration of the test was 20 minutes. After the posttest was administered, students' responses on the status of their held mental models were collected and analyzed using descriptive statistics.

Table 1

*Pretest Questionnaire*

| Select one by putting a tick (√) in the option with a correct answer. If there is no correct option, specify yours | | |
|---|---|---|
| Consider the following code fragment: –<br>*int *p;*<br>*int x;*<br>Suppose that we have the memory allocation for *p* and *x* as shown in Figure 1 below:<br><br> <br>Fig 1. Memory allocation.<br>Suppose that the following statements are executed *in the order given:*<br>*x = 50;*<br>*p = &x;*<br>**p = 38;* | After the execution of statement **p = 38*, the new values of **p, & x,* and *x* are: –<br>☐ **p = 38, &x = 1750, x = 38*<br>☐ *(*p = 50, &x = 1750, x =50)*<br>☐ **p=50, &x=1400, x= 50,*<br>☐ *(p=38, &x = 1750, x =50)*<br>☐ **p = 88, &x = 1750, x=50*<br>☐ **p = 50, &x = 1750, x=38*<br>☐ **p = 38, &x = 3150, x = 38*<br>☐ **p = 38, &x = 350, x = 38*<br>☐ Any other values of **p* and *x* are:<br>**p =* *x =* | Use this column for your rough notes, please |

Two mental model categories were proposed to analyze the status of students' mental models of pointers: the Correct Mental models (CMM) and Incorrect Mental Models (IMM). In addition, the IMM was further categorized into eight other subcategories, namely IMM1, IMM2, IMM3, IMM4, IMM5, IMM6, IMM7, and IMM8. Table 2 provides more details on the categories of these mental models.

Table 2

*Categories of Mental Models*

| MENTAL MODEL | DESCRIPTION | CATEGORY |
|---|---|---|
| **p = 38, &x = 1750, x = 38* | The value stored in variable x will be overwritten by the value assigned to a pointer **p*. The two values will thus be equal. The addresses of the reference variables will not be affected during runtime | CMM |
| **p = 50, &x = 1750 x = 50* | Assigning the value to a pointer **p* has no effect on its referenced variable x; instead, the value of the dereferenced pointer that has been assigned value will retrieve the value held by the referenced variable. The addresses of the reference variables will not be affected during runtime | IMM1 |
| **p = 50, &x = 1400, x = 50* | Assigning the value to a pointer **p* has no effect on its referenced variable x; instead, the pointer will indirectly access the value stored in variable x. however, variable x will now use the address of the pointer **p* during runtime | IMM2 |
| *p = 38, &x = 1750, x = 50* | The value in variable x will not change. The pointer **p* will hold the value assigned to it like a normal variable. The addresses of the reference variables will not be affected during runtime | IMM3 |
| **p = 88, &x = 1750, x = 50* | The value assigned to the pointer will be incremented by the value from variable *x*, but the value of variable *x* will remain unchanged. | IMM4 |
| **p = 50, &x = 1750, x = 38* | The value assigned to variable *x* and pointer **p* will be swapped. The addresses of the reference variables will not be affected during runtime | IMM5 |
| *p = 38, &x = 3150, x = 38* | The value stored in variable x will be overwritten by the value assigned to a pointer **p*. The two values will thus be equal. The addresses of the pointer and reference variable address will sum up during runtime | IMM6 |
| *p = 38, &x = 350, x = 38* | The value stored in variable x will be overwritten by the value assigned to a pointer **p*. The two values will thus be equal. The addresses of the reference variables will be subtracted from that of **p* during runtime | IMM7 |
| Others | Undefined mental models. These are models that were difficult to define and interpret | IMM8 |

### ***The experiment***

This experiment aimed to investigate the impact of combining follow-up questions and worked examples in the CeliotM PV tool on improving novice programmers' held mental models of pointers. The subjects of the study were sixty-two (62) second-year undergraduate students taking the course PMT 221 (Data Structure) at the College of Natural and Mathematical Sciences (CNMS) of the University of Dodoma. These students had already learned the basic concepts of the programming course in C++. These students voluntarily participated in this study. The protocol for the test was as follows: Students were first informed about the purpose of the study. Then, students were asked to complete the pretest questionnaire within 20 minutes. After they completed the pretest questionnaire, it was collected for analysis. The results on the status of students' held mental models can be found in the next section. After completing the pretest, students were proportionally divided into two equal groups, Group 1 (G1) and Group 2(G2), based on the status of their held mental models. Both G1 and G2 were then given a similar set of pointers exercises to practice. The G1 students used the Borland C++ compiler for practice, while the G2 students used the CeliotM PV tool. After participating in the hands-on exercises for 12 hours per week for two weeks, all students completed the posttest. The duration of the test was 20 minutes. The students were not told if they could repeat the posttest. The results of the experiment are presented in the next section. It was hypothesized that students who used CeliotM would improve the status of their held mental models of pointers better than those who used conventional C++ compilers (Borland compilers).

## Results and discussion

### ***Results***

Figure 4 shows the status of the mental models that the students were holding before applying the proposed learning approach. Figure 4 shows that of the 62 students who participated in the pretest, only 27 students (43.5%) held the CMMs of pointers, while 35 students (56.5%) held the IMMs. Figure 3 shows the status of the students' held mental models of pointers before the intervention.

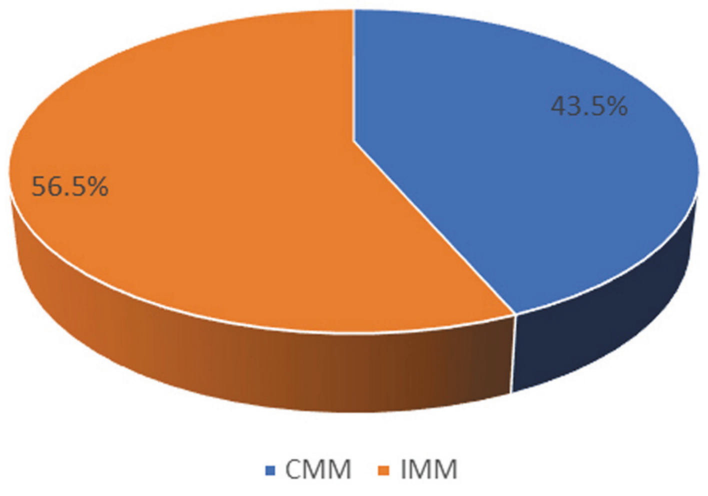


Fig. 4. The status of the students' held mental models of pointers before the intervention

As seen in Figure 4, more than half of the students (56.5%) had incorrect mental models of pointers. Table 3 shows the categories of IMMs that students had before using the proposed intervention.

Table 3

*Categories of IMMs that students had before applying the proposed approach*

| MM TYPE | N (35) | PERCENTAGE |
|---|---|---|
| IMM1 | 5 | 8.1% |
| IMM2 | 3 | 4.8% |
| IMM3 | 18 | 29.0% |
| IMM4 | 2 | 3.2% |
| IMM5 | 3 | 4.8% |
| IMM6 | 1 | 1.6% |
| IMM7 | 1 | 1.6% |
| IMM8 | 2 | 3.2% |

As shown in Table 3, the largest proportion of students with wrong mental models belonged to the IMM3 category. Such students had the misconception that assigning the value to a pointer **p* had no effect on its referenced variable *x* (**p = 38, &x = 1750 x = 50*). This misconception happened because when declaring a pointer, some students tend to confuse between the roles of memory address and variable location. Basically, when you assign a value to variable *x*, the value assigned to variable *x* will be stored to the address, say *FFFF* of variable *x*. It follows that if you assign address *FFFF to* the pointer **p* and then assign a value to the pointer **p,* the original value of variable x will be overwritten by the new value assigned to pointer **p*. The output values for both variable *x* and the pointer **p* will finally be the same.

Another category that performed poorly next to the IMM3 group was IMM1. This group constituted 8.1% of the total participants. Students with this category of wrongly held mental models had the misconception that any value that had been assigned to the pointer **p* would have no effect on its referenced variable x; instead, the value of the dereferenced pointer that had been assigned value would hold and output the value held by the referenced variable. These students are confused between variable assignments and address assignments.

Table 3 also shows that 4.8% of the students were holding IMM5. The majority of the students had the misconception that when you assign a new value to a variable (that was initially assigned a value) via a pointer, such values will be swapped. As shown from the code fragment in Table 1, since the only statement that exists between the statements in line 3 (i.e., *x = 50;*) and line 5 (i. e. **p = 38;)* is line 4 (i. e. *p = &x;*); these students might have confused the difference between the pointer' address assignment and swapping. Results from Table 3 further showed that some of the students (3.2%) were found to hold wrong mental modes that were difficult to classify.

***The results of the experiment***

Having identified a large number of IMMs in this study, it was decided to investigate whether using the newly proposed strategy that combines follow-up questions and worked examples in the PV (CeliotM) could help improve the status of students' mental models of pointers. Table 4 shows the results of the proposed approach on students held mental models of pointers between G1 (control group) and G2 (experimental group).

Table 4

*The status of students held mental models of pointers after the intervention for G1 and G2*

| MM Type | Non experimented sample (N = 62) | | G1 (N = 31) | | | G2 (N = 31) | | |
|---|---|---|---|---|---|---|---|---|
| | N | % | N | % | Change | N | % | Change (%) |
| CMM | 27 | 43.5 | 18 | 58.1 | 14.6 | 27 | 87.1 | +43.6 |
| IMM | 35 | 56.5 | 13 | 41.9 | –14.6 | 4 | 12.9 | –43.6 |

As shown in Table 4, the students who used Borland C++ compilers (G1) slightly improved their held mental models by 14.5%., while those who used CeliotM (G2) to learn pointers improved the status of their held mental models from IMM to CMM by 43.6%, raising the percentage to 87.1%. This is a substantial number taking into consideration that pointer is one of the challenging topics for novice programmers to grasp. Table 5 shows the detailed distribution of the IMMs with reference to the preliminary status of the students held mental models before the intervention.

Table 5

*Detailed distribution of the IMMs after the intervention*

| IMM TYPE | Non experimented sample (N = 62) | | G1 (N = 31) | | | G2 (N = 31) | | |
|---|---|---|---|---|---|---|---|---|
| | N | % | N | % | Change | N | % | Change (%) |
| IMM1 | 5 | 8.1 | 3 | 9.7 | −1.6 | 1 | 3.2 | −4.9 |
| IMM2 | 3 | 4.8 | 1 | 3.2 | −16.1 | 0 | 0 | −4.8 |
| IMM3 | 18 | 29.0 | 4 | 12.9 | 0 | 3 | 9.7 | −19.3 |
| IMM4 | 2 | 3.2 | 1 | 3.2 | −1.6 | 0 | 0 | −3.2 |
| IMM5 | 3 | 4.8 | 1 | 3.2 | 1.6 | 0 | 0 | −4.8 |
| IMM6 | 1 | 1.6 | 1 | 3.2 | 1.6 | 0 | 0 | −1.6 |
| IMM7 | 1 | 1.6 | 1 | 3.2 | 0 | 0 | 0 | −1.6 |
| IMM8 | 2 | 3.2 | 1 | 3.2 | 1.6 | 0 | 0 | −3.2 |

As indicated in Table 5, the IMM3 for the experimental group (G2) was largely affected by the intervention since the number of students who were holding IMMs decreased by 19.3% compared to other IMMs. It is also worth noting that whereas all IMMs were improved in G2, this was not the case for students in G1. For example, while mental model categories IMM6, IMM7, and IMM8 were all improved and eliminated for G2, they were not improved at all for G1. This implies that students the use of CeliotM helped to improve the status of the students' held mental models compared to the use of the traditional lecture method.

## *Discussion*

This study had two main objectives: to analyze the status of mental models of pointers (focusing on value and address assignment); and to evaluate the impact of combining worked-examples and follow-up questions in the CeliotM PV tool in the learning of pointers.

The study findings show that more than half of the students (56.5%) in the Data structure course had an incorrect mental model of pointer assignment. Such students were holding a wide range of incorrect mental models of pointer value and address assignment. That misconception happened because students did not exactly understand the roles and functions of the pointer. In general, a pointer is a variable. When you declare a pointer, it is given both the memory address and location, just like other variables. But unlike a normal variable, the pointer contains only the address of the variable. When you assign a value to the pointer, that pointer does not actually store that value but points to the memory location to which it was referenced.

As shown from the code fragment in Table 1, the statement *x = 50;* is followed by the statement *p = &x,* which implies assigning the address of variable x to the pointer **p*. The symbol *&* is called the addressing operator and is used in particular to pass the address of a particular variable to the pointer variable so that the pointer can point to that variable. Also, the symbol (*) that precedes the pointer identifier p is called a dereferencing or redirection operator. It is read as "value pointed to by." The function of the dereferencing operator is to restore the memory location pointed to by the value of the variable. More specifically, the dereferencing operator is used to indirectly access the data stored at the memory location pointed to by the pointer. Any change in the dereferenced pointer directly affects the value of the variable it points to.

Findings from the study have also shown that the use of the proposed approach that applies a combination of worked examples and follow-up questions in the CeliotM PV tool has managed to improve the status of students' held mental models of pointers up to 87.1%. The results suggest that using the proposed approach can greatly improve the status of students' held mental models of pointer threshold concepts. One possible reason that the proposed approach improved the status of students' flawed mental models of pointers was that, while the use of PV helped students understand the notional machine, follow-up questions helped students naturally engage in the learning process, explore deeper knowledge from their own minds and learn from others. In this way, students managed to construct and reconstruct their false mental models and hence, greatly improved their programming comprehension.

## Conclusion and future research work

In this study, eight categories of students' wrongly held mental models of pointers were identified. To improve the mental models, a strategy that involves the use of worked examples along with follow-up questions within the CeliotM PV tool was employed. The results show that the use of the proposed approach largely helped to reduce the number of students with flawed mental models of the pointers compared to the use of the traditional lecture approach. The proposed strategy seems to have a positive effect on improving students' conceptual, program tracing, and problem-solving skills. Analyzing students' mental models is very important because it allows instructors to explore the overall picture of students' comprehension patterns. It also allows instructors to identify the type (s) of student misconceptions and thus determine the best strategy to help students understand programming. This study identified several misconceptions that students have when learning pointers. However, it only examined the status of the students' held mental models of pointers within a short period of time and with a small sample size. Future studies should examine the effectiveness of using the proposed approach with a larger sample in longitudinal studies. In addition, the study focused on static pointer´s address and value assignment. It did not go so far as to try to investigate the status of students' mental models of dangling pointers, dynamic pointer declarations, and the use of pointers as reference parameters in function calls. Future studies should also investigate the status of students' held mental models with respect to these aspects.

## References

1. Hind M. Pointer Analysis: Haven' t We Solved This Problem Yet? ACM 1-58113-413-4/01/0006. 2001.
2. Kumar A.N., Road R.V. Learning the Interaction Between Pointers and Scope in. 2001;45–8.
3. Daly J. The Usage of Pointers, Arrays, and Structures. 2012;1–12.
4. Adcock B., Bucci P., Heym W.D., Hollingsworth J.E., Long T., Weide B.W. Which Pointer Errors Do Students Make? 2007;9–13.
5. Sanders K, Mccartney R. Threshold Concepts in Computing: Past, Present, and Future. 2016;91–100.
6. Lahtinen E., Ala-Mutka K., Järvinen H.M. A study of the difficulties of novice programmers. Proc 10th Annu SIGCSE Conf Innov Technol Comput Sci Educ. 2005;14–8.
7. Kumar A, Kumaresan S. Use of Mathematical Software for Teaching and Learning Mathematics. 2001:373–88.
8. Johnson-laird P.N. Mental models and human reasoning. 2001.
9. Johnson-laird P.N. Mental models and human reasoning. 2010.
10. Ma L., Ferguson J., Roper M., Wood M. Investigating the Viability of Mental Models Held by Novice Programmers. 2007;499–503.

11. Heinsen Egan M., McDonald C. Program visualization and explanation for novice C programmers. Conf Res Pract Inf Technol Ser. 2014;148:51–7.
12. Norman D.A. Some observations on mental models. Ment Model. 1983;7(112):7–14.
13. Mccauley R., Hanks B., Fitzgerald S., Murphy L. Recursion vs. Iteration: An Empirical Study of Comprehension Revisited. 2015;350–5.
14. Danielsiek H., Paul W., Vahrenhold J. Detecting and understanding students' misconceptions related to algorithms and data structures. In: Proceedings of the 43rd ACM technical symposium on Computer Science Education. 2012:21–6.
15. Winslow L.E. Programming Pedagogy – A Psychological Overview. ACM SIGCSE Bull. 1996;28(3):17–23.
16. Sorva J. Notional Machines and Introductory Programming Education. 2013;13(2).
17. Moreno A. Animation Re-designing Program Animation. University of Eastern Finland;2014.
18. Edwards-Leis C. Challenging learning journeys in the classroom : Using mental model theory to inform how pupils think when they are generating solutions. 2010:153–62.
19. Rørnes K.M. Mental Models in Programming Python. 2019.
20. Almadhoun E., Parham-Mocello J. Exploratory Study on Accuracy of Students' Mental Models of a Singly Linked List. In: 2021 IEEE Frontiers in Education Conference (FIE). IEEE; 2021:1–9.
21. Mtaho A.B. Devising a New Congruent Visualization Framework for Minimizing Learners ' Cognitive Load in Teaching a Learning Data Structures and Algorithms. The University of Dodoma; 2020.
22. Sorva J., Karavirta V., Malmi L. A review of generic program visualization systems for introductory programming education. ACM Trans Comput Educ. 2013;13(4).
23. Dehnadi S., Bornat R. The camel has two humps (working title). Middlesex Univ UK [Internet]. 2006;1–21. Available from: http://mrss.dokoda.jp/r/http://www.eis.mdx.ac.uk/research/PhDArea/saeed/paper1.pdf
24. Petre M., & de Quincey E. A gentle overview of software visualisation. PPIG News Letter, 2006:1–10. (Original work published).
25. Wright A.F., & Laboratories A. Taxonomies of Visual Programming and Program Visualization*. 1990;0:97–123. (Original work published).
26. Bergin J., Patiho-matt M., Brodlie K., Mcnally M., College A., Goldweber M., … Wilson J. An overview of visualization: Report of the Working Group on Visualization. 1996:192–200. (Original work published).
27. Scott A.S. Using Flowcharts , Code and Animation for Improved Comprehension and Ability in Novice Programming Certificate of Research. 2010 (March); 430. Retrieved from http://dspace1.isd.glam.ac.uk/dspace/bitstream/10265/460/1/Dr Andrew Scott – PhD Thesis.pdf (Original work published).
28. Moreno A., Myller N., & Sutinen E. Visualizing Programs with Jeliot 3. 2004. (Original work published).
29. Helminen J., & Malmi L. Jype – A Program Visualization and Programming Exercise Tool for Python Categories and Subject Descriptors. 2010:153–162. (Original work published).
30. Jonathan F.C., Karnalim O., & Ayub M. Extending The Effectiveness of Algorithm Visualization with Performance Comparison through Evaluation-integrated Development. 2016:16–21. (Original work published).
31. Nikander J., Helminen J., & Korhonen A. Algorithm Visualization System for Teaching Spatial Data Algorithms. Journal of Information Technology Education: Innovations in Practice, 2010;9:201–225. https://doi.org/10.28945/1305 (Original work published).
32. Hidalgo-Céspedes J., Mar\'\in-Raventós G., & Lara-Villagrán V. (2016). Learning principles in program visualizations: A systematic literature review. Proceedings – Frontiers in Education Conference, FIE, 2016-Novem. https://doi.org/10.1109/FIE.2016.7757692 (Original work published).

**Adam B. Mtaho,** PhD, Lecturer, Department of in Information and Communication Technology, Arusha Technical College, P. O. Box 296, Arusha, Tanzania.
E-mail: abasigie@.yahoo.com

*Submitted July 7, 2022*