International Journal of Computer Applications (0975 – 8887)
Volume 85 – No 9, January 2014

Using Memory Transfer Language (MTL) as a Tool for Program Dry-running

Leonard J. Mselle
Computer Science
The University of Dodoma
Dodoma, Tanzania

ABSTRACT

In this paper, the use of visualization techniques in teaching and learning programming is revisited. It is demonstrated that MTL can be used to visualize most of programming aspects. MTL, as a tool for dry-running programs, tracing and correcting codes is used in a class experiment. Results show that MTL can be used in teaching novice programmers to improve their coding abilities.

General Terms

Teaching, Learning to program

Keywords

Program visualization, Memory Transfer Language (MTL), comprehension, program dry-running

1. INTRODUCTION

Flow charts, pseudo codes and dry-running are among the traditional tools used for programming comprehension. Disciplines with a high dose of abstractness such as mathematics, physics and programming are usually taught with a combination of various tools and concrete models to simplify both teaching and comprehension. In general, it can be said that, for a subject of high complexity such as programming, the necessity for invention of various concrete models and tools for simplification of teaching, is highly demanded. For computer programming, compared with a discipline like mathematics, the number of tools and the variety of approaches is still very low. The reason why invention and application of varieties of teaching models and techniques are so much underdeveloped in programming, may be due to the reality that machine debugging and compilation have traditionally been assumed to be sufficient in taking care of the business. However, various research findings report that, teaching and understanding programming has stubbornly remained an uphill battle that does not seem to get an easy solution [1], [2].

Recent attempts to introduce visualization as a technique for teaching programming have produced promising results [3], [4], [5]. However, most of the visualization techniques are still very much entangled with machine mechanisms. In addition, visualization in general is still underdeveloped.

While machine-driven visualizations are pivotal in program debugging and comprehension, they have a negative effect of inducing hopelessness to a weak novice programmer. Perkins et al [6] report that novices often attribute human-like reasoning to the machine. This has a negative influence in debugging because when the compiler reveals bugs, a novice who does not have confidence in his debugging capabilities feels that the machine is stubbornly rejecting to understand what the novice wants the machine to understand. Researchers suggest that if students were patient enough to soft-track or dry-run their codes, they would succeed in discovering the errors and proceed to successfully produce a correct code [1], [6].

Program dry-running and flow charts are the traditional tools that are used for manual tracing of the code to verify its correctness. Each of these methods have had limited success in their application. Flow charts are used to generally represent the logic that the programmer wants to put in the machine in the form of statements. Flow charts however, lack the means to show how correct a given line of code (the syntax) is.

2. PROGRAM DRY-RUNNING

To execute a program by hand, writing values of variables and other run-time data on paper, in order to check its operation or to track down a bug is called dry-running. A dry- run is an extreme form of desk check and is practical only for fairly simple programs and small amounts of data. Most of visualization techniques rely, to a certain degree, on program dry-run which constitutes a mental run of a computer program, where the computer programmer examines the source code one step at a time and determines what it will do when run. In theoretical computer science, a dry-run is a mental run of an algorithm, sometimes expressed in pseudo code, where the computer scientist examines the algorithm's procedures one step at a time. In most cases, the dry-run is frequently assisted by a table (on a computer screen or on paper) with the program or algorithm's variables on the top. Dry-running is similar to proof reading. It is based on the assumption that the programmer knows for sure how correctly the given line should be written.

2.1 Dry-running and trace tables

By their nature, dry-run and trace tables normally can go together. Combining trace tables and dry-run evolved as a traditional tool and method for code verification. Trace tables were used for debugging and teaching programming when Pascal and FORTRAN were the teaching languages [7]. The use of trace tables for code dry-running is demonstrated in Figure 1.

Code segment

```
int x=0;
int i=1;
while(i<5){
    x=x+i;
    i++;
}
```

Trace Table

i	x
1	1
2	3
3	6
4	10

Fig 1: Dry-running a code segment using a trace table

Using the trace table, in Figure 1, the program is mentally run, and values associated with variables *x* and *i* are checked at each step of the code execution.

In programming, a given problem can be solved through more than one construct. The choice for a construct to use is a matter of convenience including mastery of one alternative over the other by the student. Consider for example, the case of *while* and *for* loop constructs. Solving the problem, as illustrated in Figure 1 with a *for* loop construct, would produce the code segment and a trace table as depicted in Figure 2.

Code segment

```
int x=0;
int i;
for (i=1;i<5; i++){
      x=x+i;
      i++;
}
```

Trace Table

i	x
1	1
2	3
3	6
4	10

Fig 2: Use of *for* instead of *while* loop construct

The trace table remains the same even though the syntax has changed. In the literature, the two constructs are said to accomplish the same thing. This is visibly demonstrated more clearly by the trace table.

For reasons not yet clear, trace tables do not feature in modern programming books, teaching notes or syllabuses. No apparent reason is given for this abandonment. In this research, a survey carried out in 56 programming books at four universities in Tanzania and Rwanda found that there was no single title that had made reference to trace tables. Since trace tables and dry-run can be used to associate the code with RAM they provide the means for learners to visibly compare the syntax of the code with the semantics of the machine.

3. HYPOTHESIS

Abstractness and general complexity have been a historical problem in the realm of teaching and learning programming. Waguespack [8] reports that poor programming skills and complete inability to write program after two or even three years' study appeared to be a common problem to most computer science students. He maintains that most computer science students graduate with weaknesses reflected in:

i. Prior knowledge in the fundamental concepts and general programming principles.
ii. Understanding of basic codes.
iii. Confidence in writing any program due to poor memory of syntax.

Similar views are expressed by Dehnadi and Bornat [2] who conclude that most students from all universities and colleges, without exception in the type of the department's intake fail the first programming course. It is hypothesized that programming abstractness can be tackled by applying visualization approach capable of mimicking computer RAM for most of programming aspects. Such a visualization technique must satisfy the following conditions:

i. Be consistently and invariably applicable in all basic programming aspects, i.e. variable declaration, data feeding, data output, flow of control, functions, arrays and file handling.

ii. Be independent of programming languages, i.e. it must be capable of being applied in teaching any programming language without changing its substance.

iii. Be machine independent, i.e. it can be applied effectively with or without a machine. That is, it can be absolutely manual (paper-and-pencil) program analyzer/builder and debugger, without denying it the possibility of creating a machine-based version whenever it is desired.

iv. Be useful for mentally visualizing and verifying correctness or incorrectness of the code line-by-line, and provide a lead for a possible correct solution.

v. In addition, it must provide features itemized by Ramadhan and Du Boulay [9] in their DISCOVER system. That is to say, a novice can use it to conceptualize the solution path and direction. So, it must visualize the dynamic behavior of programs in relation to machine and it must be able to provide the programmer with feedback on a successful step towards a solution.

4. REVIEW OF PROGRAM VISUALIZATION TECHNIQUES

Putnam et al [1] found that students had difficulty in keeping track of the values of variables when tracing programs. This constituted an obstacle in writing correct programs and in debugging. Numerous researchers have come to a similar conclusion that the big obstacle for novice programmers, seems to be the early misconception about variables [6], [10], [11] and [13].

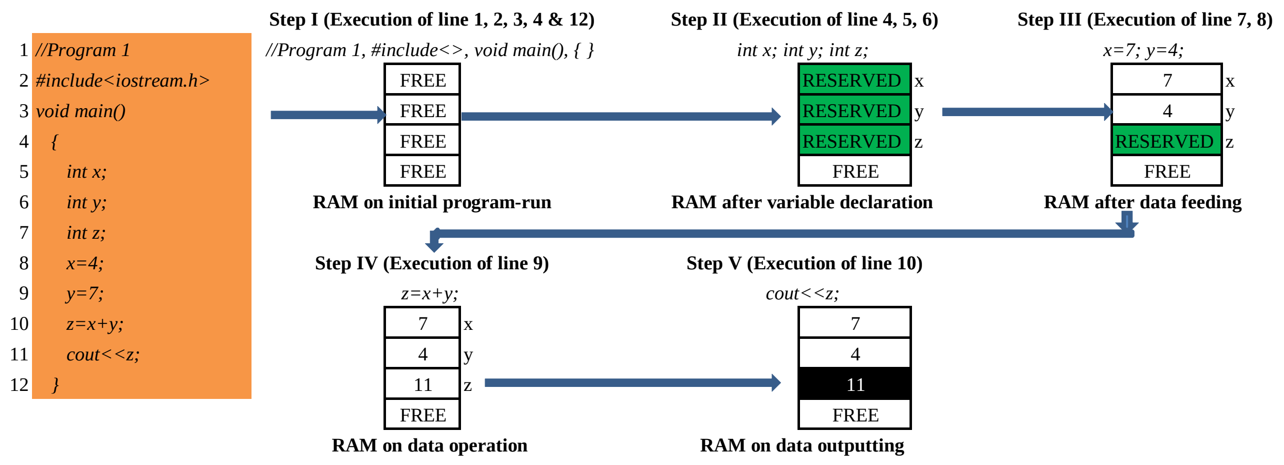
Advocates of program visualization have reported progress in increasing programming comprehension by introducing animation tools in teaching programming. Scott et al [12], Ala-Mutka [3], Ben Ari [5] and Ramadhan and du Boulay [9] are among those who have confirmed that visualization can increase programming comprehension. Animation tools have the ability to show the novice what is happening inside variables in the machine; as a result comprehension is enhanced.

In spite of their experimental success, animation tools have yet to find popular use in the realm of teaching programming. They do not feature in mainstream programming books [13]. This possibly may be due to their infancy or due to the fact that they are still entangled with machine mechanisms, which taint them with a parochial character.

Although some of visual tools such as DISCOVER [9] which combine both intelligent and unintelligent approaches are available, MTL, as depicted in Figure 3 through 7, is made to be applied outside the machine in order to be transformed into a general, rigid, completely machine-independent tool which can be applied to any programming language, outside the machine environment, without denying it the flexibility to be developed into machine-driven versions, if one wishes to.

5. USING MTL FOR PROGRAM DRY-RUNNING

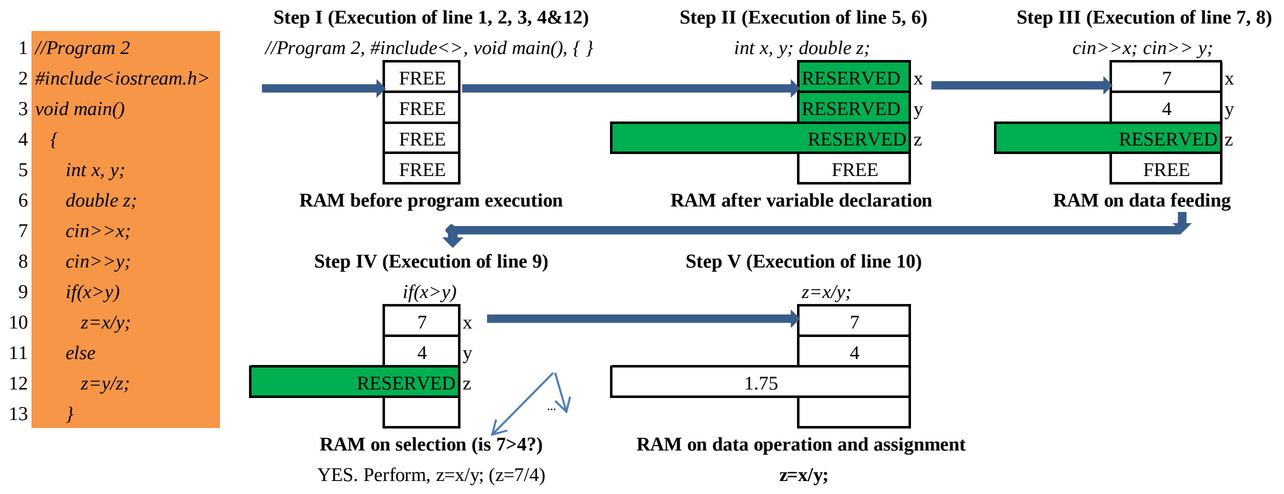
MTL, as a modified version of trace tables, constitutes a visualization tool that can completely rely on paper and pencil without eliminating a soft version in its design. Computer RAM is used as the only concrete model to represent the notional machine. The source-code only becomes relevant when its meaning is extrapolated with its effect in the RAM, line by line. RAM diagrams (modified trace tables) allow a novice to close track the code by visualizing its execution in the RAM. It allows the novice programmer, using paper and pencil, to substitute the code, line by line, in the computer memory and find out if the meaning of the code, as written by the programmer, matches with the meaning of the code as is supposed to be understood by the computer. RAM diagrams, as components of MTL, are demonstrated in Figure 3 through 7.



```
 1 //Program 1
 2 #include<iostream.h>
 3 void main()
 4  {
 5    int x;
 6    int y;
 7    int z;
 8    x=4;
 9    y=7;
10    z=x+y;
11    cout<<z;
12  }
```

Step I (Execution of line 1, 2, 3, 4 & 12)
//Program 1, #include<>, void main(), { }
FREE
FREE
FREE
FREE
RAM on initial program-run
Step II (Execution of line 4, 5, 6)
int x; int y; int z;
RESERVED x
RESERVED y
RESERVED z
FREE
RAM after variable declaration
Step III (Execution of line 7, 8)
x=7; y=4;
7 x
4 y
RESERVED z
FREE
RAM after data feeding
Step IV (Execution of line 9)
z=x+y;
7 x
4 y
11 z
FREE
RAM on data operation
Step V (Execution of line 10)
cout<<z;
7
4
11
FREE
RAM on data outputting

Fig 3: Using MTL to dry-run/close-track variable declaration, data inputting, data processing, data outputting and SEQUENCE



```
 1 //Program 2
 2 #include<iostream.h>
 3 void main()
 4  {
 5    int x, y;
 6    double z;
 7    cin>>x;
 8    cin>>y;
 9    if(x>y)
10       z=x/y;
11    else
12       z=y/z;
13    }
```

Step I (Execution of line 1, 2, 3, 4&12)
//Program 2, #include<>, void main(), { }
FREE
FREE
FREE
FREE
RAM before program execution
Step II (Execution of line 5, 6)
int x, y; double z;
RESERVED x
RESERVED y
RESERVED z
FREE
RAM after variable declaration
Step III (Execution of line 7, 8)
cin>>x; cin>> y;
7 x
4 y
RESERVED z
FREE
RAM on data feeding
Step IV (Execution of line 9)
if(x>y)
7 x
4 y
RESERVED z
RAM on selection (is 7>4?)
YES. Perform, z=x/y; (z=7/4)
Step V (Execution of line 10)
z=x/y;
7
4
1.75
RAM on data operation and assignment
z=x/y;

Fig 4: Using MTL to dry-run/close-track variable declaration, data inputting, data processing, data outputting, SEQUENCE and BRANCHING

```
 1 // Program 3
 2 #include<iostream.h>
 3 void main()
 4 {
 5   int sq(), x, z;
 6   cout<<"Enter a number";
 7   cin>>x;
 8   z=sq(x);
 9   cout<<"The square of"<<x;
10   cout<< "is"<<z;
11 }
12   int sq(y)
13   {
14     return(y*y);
15   }
```

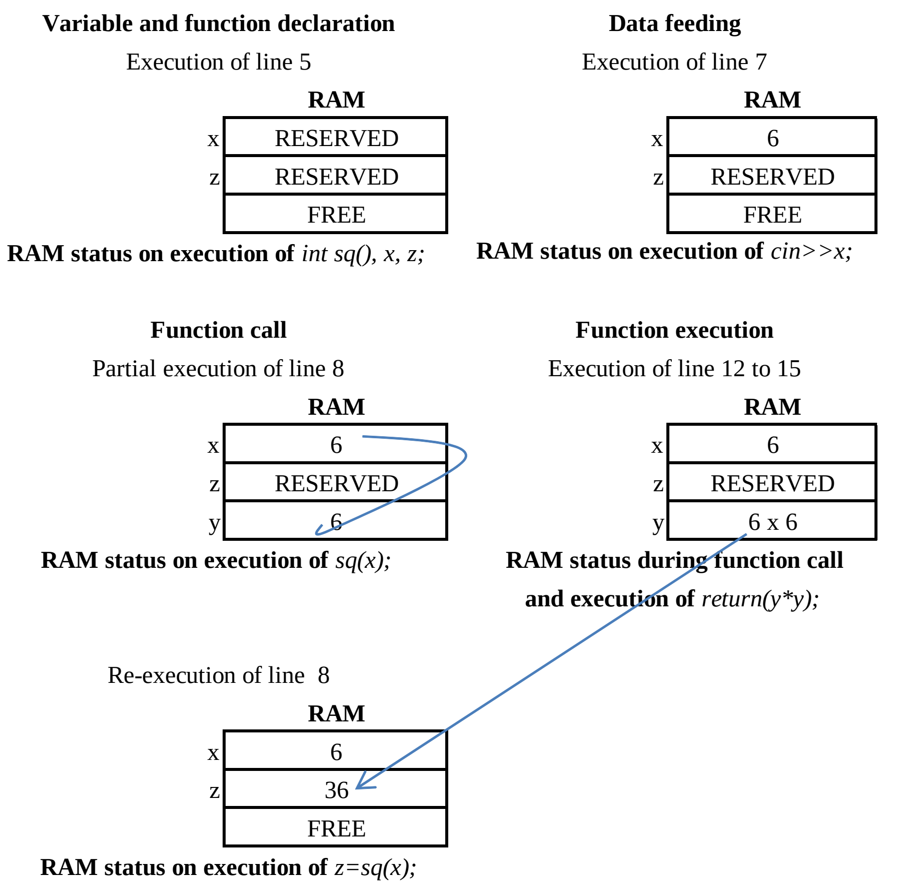

Variable and function declaration
Execution of line 5
RAM
x RESERVED
z RESERVED
FREE
RAM status on execution of int sq(), x, z;
Data feeding
Execution of line 7
RAM
x 6
z RESERVED
FREE
RAM status on execution of cin>>x;
Function call
Partial execution of line 8
RAM
x 6
z RESERVED
y 6
RAM status on execution of sq(x);
Function execution
Execution of line 12 to 15
RAM
x 6
z RESERVED
y 6 x 6
RAM status during function call and execution of return(y*y);
Re-execution of line 8
RAM
x 6
z 36
FREE
RAM status on execution of z=sq(x);

Fig 5: Using MTL to dry-run/close-track variable and function declaration, data inputting, data processing, SEQUENCE and FUNCTION CALL and PARAMETER PASSING

```
 1 //Program 4
 2 #include <iostream.h>
 3 void main()
 4 {
 5   int z[4];
 6   cin>>z[0];
 7   cin>>z[1];
 8   z[2]=8;
 9   z[3]=400;
10 }
```

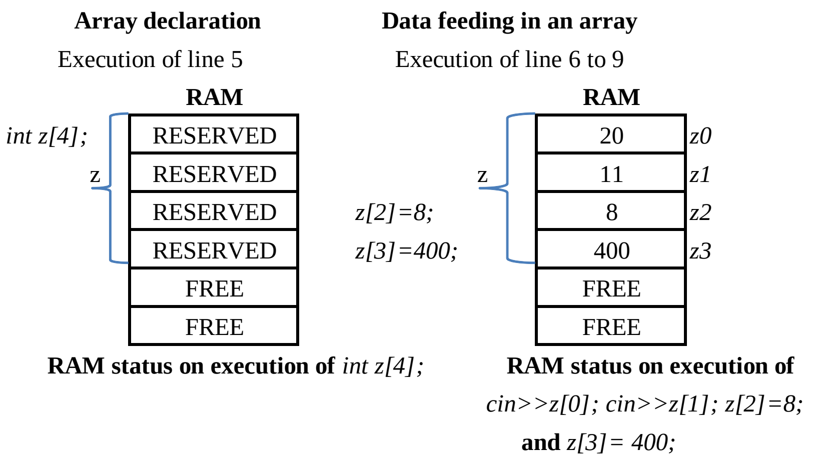
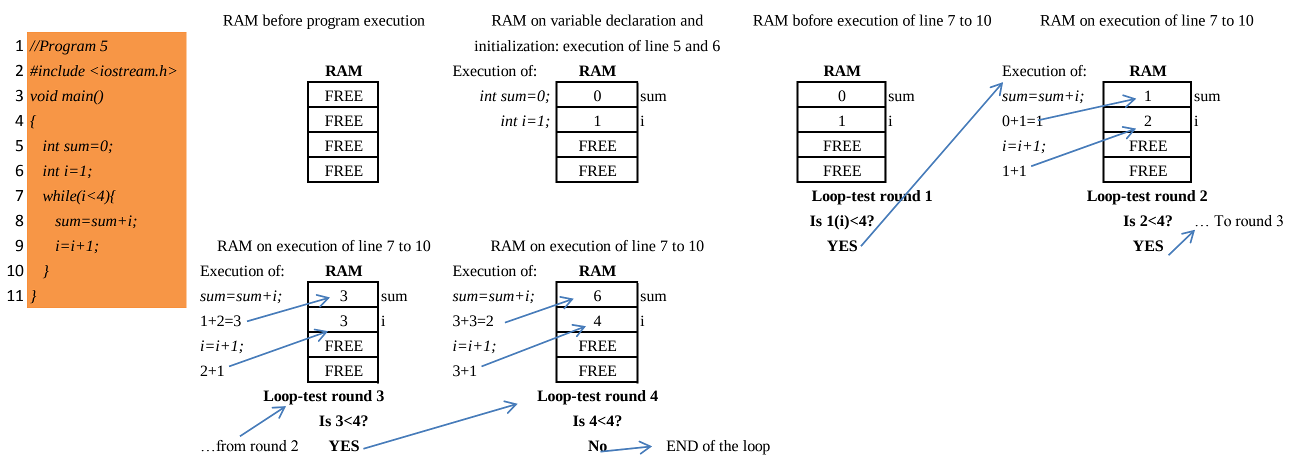


Fig 6: Using MTL to dry-run/close-track array declaration and data feeding in an ARRAY



```
1  //Program 5
2  #include <iostream.h>
3  void main()
4  {
5    int sum=0;
6    int i=1;
7    while(i<4){
8      sum=sum+i;
9      i=i+1;
10   }
11 }
```

Fig 7: Using MTL to dry-run/close-track variable declaration, variable initialization and LOOPING

As demonstrated in Figure 3 and Figure 4, MTL, relying on rectangles, enables the learner to mimic the computer RAM for visualizing variable declaration, data inputting, data processing, data outputting, SEQUENCE and BRANCHING. In Figure 5, the concept of cooperation between two functions (function call and parameter passing) is visualized using the same mechanism of rectangles. Figure 6 shows how arrays are declared and inputted with data. Similarly, MTL can mimic how data from arrays can be outputted by employing rectangles to represent RAM. Figure 7 demonstrates how MTL achieves visualization of LOOPING using the same style of RAM mimicry by rectangles.

Conceptually, it can be said that MTL, as demonstrated in Figure 3 through 7, can be applied consistently and invariably in all basic programming aspects i.e. variable declaration, data feeding, outputting, flow of control, functions, arrays and file handling. MTL is independent of a programming language, i.e. it can be applied in teaching any programming language without changing its substance. MTL is machine independent, i.e. it can be applied effectively with or without a machine. That is to say, it can be absolutely a manual (paper-and-pencil) program analyzer/builder and debugger, without denying the possibility of creating a machine-based version whenever it is desired. It can be used to mentally visualize and verify correctness or incorrectness of the code, line by line, and provide a lead for a possible correct solution. In addition, MTL can be used by a novice to conceptualize the solution path and direction, visualizing the dynamic behavior of programs in relation to the machine while providing a programmer with feedback on a successful step towards a solution.

6. THE EXPERIMENT

6.1 The study setting

To test the effectiveness of MTL as a tool for tracking, dry-running and detecting program correctness/incorrectness, 156 second-year students, all computer science majors, were involved in the experiment.

The group had spent 162 hours learning programming as follows: C++ Programming (for a total of 54 hours in the first semester), Data Structures and Algorithms (for a total of 54 hours in the second semester) and Visual Basic (for 54 hours in the second year’s first semester). Teaching had been carried out by combining lectures, tutorials and laboratory classes.

When the group was studying Operating Systems during the second semester in the second year, before engaging in the topic of processes, an elementary programming quiz as shown in Figure 8 was administered, as a means to revise basic programming knowledge, as a preparation for studying the concept of processes in operating systems.

Quiz: Time allowed 6 minutes.

Given that 1 yard is equal to 0.914 meters,

write a code to convert 6 yards into meters.

Model solution

```
#include<iostream.h>
void main()
{
  int yards=6;
  double meters;
  meters= 0.914*yards;
  cout<<” 6 Yards =  "<<meters <<"  Meters";
}
```

Fig 8: Quiz number 1 and the model answer

Students were given six minutes to write the code. After collecting the scripts, they were analyzed to find out correct and incorrect answers. The count revealed that only 13 candidates out 156 were able to write correct codes. Answers from incorrect codes were analyzed to find out the type of errors committed. These were distinctively grouped in three categories as:

i. Reference to undeclared variables; i.e. *int yards, float meters, yards6=yards*meters;* (variable *yards6* is not declared).

ii. Multiple data feeding; i.e. *int yards=6; cin>>yards;* (variable *yards* is referenced both in the assignment and in the *cin>>* input stream).

iii. Data type mismatch; i.e. *int yards, meters; meters=yards*0.914;* (variable *meters* is declared as **integer** but it is assigned a **float** value).

These errors and their frequencies are summarized in Table 1.

Table 1. Distribution of errors before MTL

Reference to undeclared variables	Multiple data feeding	Data type mismatch	Others
63	60	33	22

On analyzing a sample of 160 past examination scripts, it was revealed that most students conceived program codes as solid facts like history or geography. They considered codes to be static and discrete. A big proportion of students performed poorly in any question that required application of general concepts. Most students had passed because they had memorized some codes which had been discussed earlier during class sessions and had reappeared in tests and in the examination. One can compare this with the case of mathematics students memorizing examples of mathematical problems, perceiving them as all about the subject and succeeding to do the question, only if it had earlier been discussed. These findings confirm those by Waguespack [4], and Dehnadi and Bornat [2] who contend that most computer science candidates graduate as non programmers. It is further confirmed that teaching programming for many hours and teaching students multiple languages is a waste at best and cruel at worst if the fundamentals are not firmly mastered from the beginning [1], [2].

6.2 Assessing the impact of MTL

As a refresher, four hours were set aside for revision. Throughout the revision, complex programming aspects such as loops, and files were re-introduced to students. The entire discussion was carried out by the aid of MTL. In the end of revision, a question, as depicted in Figure 9, was administered. In addition, students were instructed to demonstrate their inputs and outputs using MTL.

Quiz: Time allowed 6 minutes.

Given two numbers (x and y) write a code to perform division operation on them, and store the inputs and the output on a file in the disk

Model solution

```
#include<iostream.h>
#include<fstream.h>
void main()
{
  int x, y;
  ofstream savefile;
  savefile.open(“Myfile.txt”);
  cout<< “Enter the numerator”<<’”\n”;
  cin>>x;
  cout<< “Enter the denominator ”<<’”\n”;
  cin>>y;
  savefile<<x<<”\n”;
  savefile<<y<<”\n”;
  savefile<<x/y;
  savefile.close();
}
```

Fig 9: Quiz number 2 and the model answer

7. RESULTS AND DISCUSSION

When all 156 examination scripts were analyzed for errors, the distribution of errors was as summarized in Table 2.

Table 2. Distribution of errors

Reference to undeclared variables	Multiple data feeding	Data type mismatch	Others
8	4	9	11

Analysis of errors revealed that all those who had wrong answers had not been able to employ MTL correctly in tracing their codes.

Percentage of errors committed before and after introduction of MTL is summarized in Table 3 and sketched in Figure 10.

Table 3. Percentage of errors committed by students before and after the use of MTL

Reference to undeclared variables		Multiple data feeding		Data type mismatch		Others	
Before	After	Before	After	Before	After	Before	After
0.543	0.068	0.517	0.034	0.284	0.077	0.189	0.094

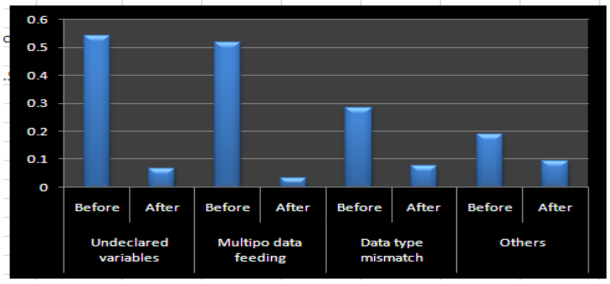


Fig 10: Percentage of errors committed before and after introduction of MTL

Davies [11] posits that understanding the variables, their relationship and roles is the core to understanding programming. Early elimination of misconceptions about variables is the cornerstone to understanding programming. With a substitution tool like MTL, novices can avoid making reference to undeclared variables, and multiple data feeding since every variable visibly changes the status as it gets and/or exchanges values. Confusion about data types is minimized since each value inside a variable is seen and can be checked whether it corresponds to its type-size or not.

8. CONCLUSIONS AND FUTURE WORK

MTL is tailored to ensure that each code-line is explained and justified with its corresponding impact on the RAM. The impact of each code-line on the RAM is diagrammatically visualized by depicting the perceived code-line impact or the value in the corresponding cell of the RAM. The use of MTL to close track the code, suppresses the possibility for students to attribute computers with human reasoning abilities; a concern that was pointed out by Perkins et al [6]. Using MTL the learner is in absolute control of the process. According to Dehnadi and Bonart [2], the inability to attribute meaning to the code is the source for novices giving up programming. MTL provides a means for a programmer to replay the rules that the machine follows to get the results. It enables the programmer to see the meaning that each code is making to the machine. It is argued by Naps et al [13] that visualization has not been widely applied in programming because it cannot be integrated in programming books. MTL, as demonstrated in Figure 3 to 7 can be integrated in elementary programming books. MTL is suitable for most of the elementary programming aspects.

Despite these results, MTL cannot be used for visualizing big programs. MTL has never been tested in recursion and problem solving aspects. Results of this experiment cannot be used to conclude that MTL can so greatly enhance understanding of programming. The questions involved are very simple, and mostly similar. There is a need for further investigation in higher programming aspects such as recursion and problem composition.

However, the study confirms that most programming students progress to high level without having minimum knowledge about programs and programming. Elementary basics such as variables, data inputting, data processing and outputting are taken for granted as simple issues that all students can understand. However, if these aspects are not re-emphasized using concrete models such as MTL, a big number of students leave colleges without the capability for elementary programming. This state of affair, apart from the lack of effective teaching tools, can be attributed to the misguided quality assurance procedures and the efforts of colleagues who doggedly believe in normal cave [2]. It was found that most students find new concepts such as variables too complex to be grasped in the short time that they are supposed to learn and use it in programming. To mitigate this catastrophic situation, tools like MTL which can be employed in combination with other concrete tools used for teaching programming, could provide a remedy.

Future work will be directed in incorporating aspects such as recursion, library calling and problem composition in MTL.

9. REFERENCES

[1] Spohrer J. C. and Soloway, E. 1986. Some Difficulties of Learning to Program. In Soloway, E. and Spohrer,J. C. editors, Studying the Novice Programmer, pages 283–299. Lawrence Erlbaum Associates, Hillsdale, NJ, 1989.

[2] Dehnadi, S. and Bonart, R. 2006. The Camel has Two Humps (working title). School of Computing, Middlesex University, UK.

[3] Ala-Mutka, K. 2003. Problems in Learning and Teaching Programming: A Literature Study for Developing Visualizations in the Codewitz-Minerva Project. http://www.cs.tut.fi/~edge/literature_study.pdf. [Accessed on 27-11-07].

[4] Kuittinen, M., Tikansalo, T. and Sajaniemi, J. 2008, "A study of the Development of Students' Visualizations of Program State During an Elementary Object-Oriented Programming Course", ACM Journal of Educational Resources in Computing, 7(4).

[5] Ben-Ari, M. and Sajaniemi, J. 2004. Roles of Variables as Seen by Computer Science Educators. ITiCSE 2004, 52-56. [Accessed on 02-09-08].

[6] Perkins, D. N., Hobbs, H. R, Martin, F. and Simmons, R. 1986. Conditions of Learning in Novice Programmers. In Soloway, E. and Spohrer and J. C., editors, Studying the Novice Programmer, Lawrence Erlbaum Associates, Hillsdale, NJ, 1989. p. 261–279.

[7] Benedict J. H. and du Boulay, B. 1986. "Some difficulties of learning to program", Journal of Educational Computing Research, 2(1) p. 57–73.

[8] Waguespack, Jr. L. J. 1989. Visual metaphors for Teaching Programming Concepts. ACM SIGCSE Bulletin, v. 21, n. 1, p.141-145.

[9] Ramadhan, H. and Du Bolay, B. 1992. DISCOVER: Programming Environment for Novices. COMPSAC '92. Proceedings, Sixteenth Annual International Chicago, p. 375 – 380.

[10] Samurcay, R. 1989. The Concept of Variable in Programming: its Meaning and Use in Problem-Solving by Novice Programmers. In: Studying the novice programmer, Hillsdale, NJ, 1989, p.161-178.

[11] Davies, S. P. 1993. "Models and theories of Programming Strategies", International Journal of Man-Machine studies, 39 (2), p. 237-267.

[12] Scott, A., Watkins, M. and Duncan, M. 2005. A Step Back from Coding – An Online Environment and Pedagogy for Novice Programmers,: http://www.ics.heacademy.ac.uk/events/jicc11/scott.pdf. [Accessed on 02-09-08].

[13] Naps, T., R¨oßling, G., Almstrum, V., Dann, W., Fleischer, R., Hundhausen, C., Korhonen, A., Malmi, L., McNally, M., Rodger, S. and Vel´azquez-Iturbide, A. 2003. Exploring the Role of Visualization and Engagement in Computer Science Education. *ACM SIGCSE Bulletin*, 35(2), p. 131–152.