Journal of Computer Science 5 (8): 608-613, 2009
ISSN 1549-3636

Memory Tracing

Eva Cogan and Chaya Gurwitz
Department of Computer and Information Science,
Brooklyn College, Brooklyn, New York, 11218 USA

Abstract: Problem statement: Students completing introductory computing courses did not know how to program at the expected level. Seeking the underlying problem, we came to believe that students were focusing only on results and not connecting with the inner workings of their code. This left them poorly prepared to master increasingly complex problems. **Approach:** We hoped that by promoting memory tracing as a core competence as early as possible in introductory programming courses we would hone the understanding and skills of our students and improve their chances for succeeding in computer science. We emphasized a basic and manual approach to memory tracing--in the classroom, in conjunction with homework assignments and on exams--to help our students gain the ability to write good programs, test them and, should it become necessary, debug them. **Results:** Having received gratifying results from our approach in our own classes, we had moved to get the word out as quickly as possible to motivate other educators to implement it. We described how we derived benefit from memory tracing in the various contexts and we presented the details of our method for teaching students how to best use this technique. **Conclusion/Recommendations:** Training students early on to actively carry out a manual memory trace of programs (as opposed to relying on debuggers or print statements) will help them develop their coding skill and comfort, quite apart from any facility for finding and fixing errors. Although experienced programmers trace intuitively, beginning students do not; they need to be trained. Therefore we felt that tracing should be an explicit, emphasized component of the introductory courses.

Key words: Memory tracing, tracing, introductory programming, novice programming, computer science education

INTRODUCTION

Memory tracing is the process of recording the value changes of program variables. It has also been called desk checking or playing computer. This is distinct from hand-checking, doing the calculations to solve the problem independently of the program and comparing these results to the program's final output.

Although memory tracing is universally recognized as an essential skill in program debugging, it seems to us that it is not sufficiently emphasized as an effective pedagogic technique in introductory programming courses. Experienced programmers often trace intuitively. However, beginning students do not and so they need to be trained. Therefore we feel that tracing should be an explicit, emphasized component of the course. As noted in[1]: "We discovered that many students with a good understanding of programming do not acquire the skills to debug programs effectively and this is a major impediment to their producing working code of any complexity. Skill at debugging seems to increase a programmer's confidence and we suggest that more emphasis be placed on debugging skills in the teaching of programming".

The results of a multi-national study[4] support our view that teaching methods which emphasize memory tracing lead to greater student success:

> Soloway[9] claims that… skilled programmers carry out frequent "mental simulations", of both abstract designs-in-progress and code being enhanced, as a check against unwanted dynamic interactions between components of the system. He argues that such simulation strategies should be taught explicitly to students. Many of our teaching traditions date back to the era of punch cards. In the days of overnight batch runs, there was little need to explicitly encourage students to carefully check their code before submitting it for a batch run, as a careless error could waste a

Corresponding Author: Eva Cogan, Department of Computer and Information Science, Brooklyn College, Brooklyn, New York, 11218 USA

> whole day. In an era where the next test-run is only a mouse-click away, we need to place greater explicit emphasis on mental simulation as part of the process of writing code. When faced with a piece of code to read and understand, experienced programmers frequently "doodle". That is, they draw diagrams and make other annotations as part of determining the function of the code…Students were given "scratch" paper upon which they were allowed to draw pictures or perform calculations as part of answering the MCQs) Multiple Choice Questions. … Not surprisingly … if a student carefully traces through the code … thus documenting changes in variables, the likelihood of getting the correct answer is high

Similarly, in[5] it is reported that novice programming students who use annotations such as tracing perform better on multiple choice tests. In[10] we are advised, "educators should also stress that they do not use this technique for demonstration purposes only but because mental tracking of values of several variables is doomed to fail due to the limitations of human cognition".

When teaching students how to debug their programs, there are three techniques that are commonly used: employing an interactive debugger, inserting temporary print statements into the program and performing memory tracing by hand.

Most IDE's include debuggers, which are of tremendous benefit in the development of large, complex programs. Nevertheless, these interactive debuggers are not necessary for the short routines we use with novice programmers and the need to master them early on is an additional obstacle for some students. Indeed, we teach children to add and subtract before we provide them with a calculator and we teach spelling despite the advent of electronic spellcheckers. Likewise, we should train students to become proficient at doing manual memory tracing before we move them on to automated debuggers.

"An interactive debugger is an outstanding example of what is not needed-it encourages trial-and-error hacking rather than systematic design"[7]. The "habitual use of symbolic debuggers also tends to discourage serious reflection on the problem. It becomes a knee-jerk response to fire up the debugger the instant a bug is encountered and start stepping through code, waiting for the debugger to reveal where the fault is"[3].

Similarly, the technique of adding debug print statements, while it has the benefit of being simpler to learn, still does not train the student to actually work through a program.

Therefore, we want our students to manually trace the execution of a program, actively recording the changes in the variables, rather than passively observing the computer doing it. Essentially, we want the student to be the computer.

A multi-national study[6] found that students completing introductory computing courses do not know how to program at the expected skill level. A more recent study[2] also found that the student success rates in introductory programming classes are very low.

We maintain that student dissatisfaction, stemming from unrealistic expectations, contributes to this problem. The tools available to students prior to their first college experience with computer science make programming look like magic. Students expect to be able to "drag and drop" solutions to any problem and therefore have a hard time getting "into" programming. They tend to skip over implementation details in their expectation of rapidly producing fantastic results. We must keep warning our students to trace what the program actually does and not what they wish it would do. Students must realize that correct syntax and successful compilation are not sufficient. They must figure out how execution affects the variables. We feel that by emphasizing tracing early on, we shift the focus from results (where we cannot hope to compete with games and simulated universes that provide instant gratification) to process. Once students realize that in our class the process itself is a primary goal, they are less frustrated by the technicalities of programming, especially when applied to apparently trivial problems. Thus, we use this debugging technique to help students get over this initial hurdle, until they become proficient enough to take on more challenging and satisfying programming assignments.

Finally, stressing memory tracing as a means of understanding the problem to be addressed by a program, as opposed to being a drudgery of last resort invoked only after things have gone horribly and inexplicably wrong, should develop in our students a receptive attitude towards "test-driven development". A proclivity to test first leads to better understanding of functionality and improved code quality. Careful consideration of how test cases would be validated before writing the actual code leads to an understanding of what the program should do and how to approach coding it.

Approach: For all of the above reasons, emphasizing memory tracing-in the classroom, in conjunction with homework assignments and on exams-helps our students learn how to write, test and debug computer programs.

Class: We introduce students to this critical skill by tracing new programs in class, modeling the technique and helping them understand the programs. As noted in[4]:

> Even when our principal aim is to teach students to write code, we require students to learn by reading code. … We typically place example code before students, to illustrate general principles. In so doing, we assume our students can read and understand those examples…. Perkins[8] claim that the ability to perform a walkthrough is an important skill for diagnosing bugs and therefore the ability to review code is an important skill in writing code

Manually performing a memory trace makes the process concrete. Requiring students to tell the instructor the values in the memory trace encourages them to actively participate in the class.

Before writing the code for a program we develop during a lecture, we supply sample test data and determine the desired output. Directing attention to the output helps students understand what the code is supposed to do in a more concrete manner than a simple verbal description of the task. When we develop code in class, we try different approaches, tracing each in turn to see if it works. For example, when developing a "structured read loop", we first try a few (wrong) placements of the input statement. Once the class completes writing code it deems to be correct, we trace it to see exactly how the code succeeds in accomplishing the goal.

After tracing together with the instructor, students are asked to produce on their own a memory trace and output for short program segments. This allows the students to test their understanding and practice the technique. Sometimes students collaborate to correct each other's results; the whole class gains from such teamwork.

Homework: We emphasize tracing during homework as well. In addition to working through memory traces for given programs, the students are generally asked to submit a memory trace along with the programs they are assigned to develop. They are required to run their program on a small data sample that exercises various paths in the program and provide a full trace.

When students have questions about their assignments, they are required to work out a memory trace on paper before bringing their questions to the instructor. It is better for them to try the trace on their own first in order to develop confidence in their own ability. Of course, sometimes they still need help in finding an error. Tracing a fellow student's program seems to motivate students to gather around the instructor's desk to help. Again, individually and as a group, they are actively involved in the debugging, not just passively observing. Finally, the code a student brings to the instructor sometimes contains patches inserted by a tutor or friend. The student may not know why the code was inserted or what it is supposed to do. Tracing forces the program owner to figure out the role of every statement in the program.

Exams: On exams, there are three kinds of questions that involve tracing:

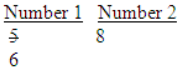
- We supply correct code and require the students to produce a memory trace and output.
- We supply incorrect code and require the students to locate and correct the bugs by tracing through the program.
- We supply a specification along with sample input and output and require the students to write the program. We recommend that they trace the program to see what it actually does. Thus they debug and correct their program before submitting it. Again, students learn to differentiate between what they hoped a program would do and what it actually does.

MATERIALS AND METHODS

We use the following technique to train students to trace various elementary programming constructs.

Simple variables: At the beginning of the course, we introduce the method of memory tracing by representing each variable as a rectangular memory cell with the sequence of its values recorded inside. The variable name is written above the rectangle:

As the semester progresses and we deal with programs in which the values of variable values change more frequently, it becomes difficult to track the change of values within a small rectangular cell. Therefore, we switch our display technique and represent the values of each variable as a vertical list under an underlined heading of the variable name:



We suggest recording a value only when it changes:

The code:
x = 0;
y = 6;
x = 7;
y = 9;
may be traced as:

x	y
~~0~~	~~6~~
7	9

Some students prefer to rewrite a snapshot of all the variables whenever any variable changes:

x	y
~~0~~	6
7	~~6~~
7	9

In small programs this does no harm, but in larger programs, it can become a nuisance.

Sometimes it is helpful to skip lines to reflect when values change. Nasty bugs caused by one variable being changed too early or too late in relation to another variable become evident when using this technique:

The code:
x = 0;
y = 6;
y = 9;
x = y - 2;
would be traced as:

x	y
~~0~~	
	~~6~~
	9
7	

Failure to declare or initialize a variable is a common error. A memory trace highlights any attempt to either assign a value to a nonexistent "memory cell" or use a nonexistent value.

Loops: We try to record each iteration of a loop on one line. We include the last unused value of the loop control variable since the variable does take on that value, even though the loop is not executed with that value:

The code:
```
sum = 0;
for (int i = 1; i <= 3; i++)
        sum += 5;
```
is traced as:

i	sum
	~~0~~
~~1~~	~~5~~
~~2~~	~~10~~
~~3~~	15
4	

Nested loops: Skipping lines in the column for the outer loop's control variable is crucial when tracing nested loops. The goal is to illustrate that for each iteration of the outer loop, we do all iterations of the inner loop:

The code:
```
for (int i …)
    for (int k…)
```
is traced as:

i	k
~~1~~	~~1~~
	~~2~~
	3
2	~~1~~
	~~2~~
	3

Functions: For many students, the hardest part of learning how to program with functions is following how parameters are passed. Tracing helps the students understand the process by producing a visual record of which values are assigned to which parameters.

We completely rewrite from scratch the trace of a function each time it is called, reflecting the actual duration of its variables. Each parameter and local variable is listed horizontally under the name of the function. In the following example, we use the same names n and sum for the identifiers in the calling function and the called function to point out that they are distinct and independent.

The code:
```
int computeSum(int);
int main()
{
    int n=3, sum;
    cout << "n is " << n << endl;
    sum = computeSum(n);
    cout << "sum is " << sum << endl;
    cout << "n is still " << n << endl;
    return 0;
}
```

```
int computeSum(int n)
{
    int sum = 0;
    for (; n > 0; n--)
        sum += n;
    cout << "In computeSum n is" << n <<  endl;
    return sum;
}
```

would be traced as:

main		computeSum	
n	sum	n	sum
3	6	~~3~~	~~0~~
			~~3~~
		~~2~~	~~5~~
		~~1~~	6
		0	

Reference parameters are more confusing than value parameters. To distinguish between them, students are cautioned not to record a value for a reference parameter, but rather to record the address of the corresponding actual parameter. We do not use an actual numeric address, but rather utilize the address symbol '&'. When students find the '&' in their trace lists, they are reminded to record the change in the namespace of the calling function, rather than in the called function. This eliminates a major tracing error:

The code:

```
void swap(int &a, int &b);
int main()
{
     int x = 5, y = 3;
     cout << "before " << endl << "x is ";
     cout << x << " , y is " << y << endl;
     swap(x, y);
     cout << "after " << endl << "x is ";
     cout << x << " , y is " << y << endl;
     return 0;
}

void swap(int &a, int &b)
{
     int temp;
     temp = a;
     a = b;
     b = temp;
     return;
}
```

is traced as:

main		swap		
x	y	a	b	temp
~~5~~	~~3~~	&x	&y	5
3	5			

Table 1: Vertical array representation

	Array name		
[0]	~~1~~	~~2~~	3
[1]	~~6~~	~~9~~	0
[2]	~~1~~	2	

Table 2: Horizontal array representation

Array name		
[0]	[1]	[2]
~~1~~	~~6~~	~~1~~
~~2~~	~~9~~	2
3	0	

Arrays: The elements of the array are listed vertically under its name, with the index on the left. The list of values for each element is horizontal (Table 1).

Alternatively, the array elements could be listed horizontally with the changes appearing in vertical columns (Table 2).

While the latter approach is more consistent with strings and allows for temporal synchronization with simple variables, students seem to find the former arrangement more intuitive.

Strings: We write the value stored in a string horizontally. The changes are listed vertically. Although this is different from the way we trace arrays, it is more natural, since in English we write from left to right.

String name		
[0]	[1]	[2]
~~e~~	~~a~~	~~t~~
d	o	g

Bubble sort: In class we develop the following code for Bubble Sort:

```
void bubbleSort(int numb[], int n)
{
    int temp, pass=0;
    bool swapped;
    do {
        pass++;
        swapped = false;
        for(int i = 0; i < n-pass; i++)
            if (numb[i] > numb[i+1]){
                swap(numb[i], numb[i+1]);
                swapped = true;
            }
        } while (swapped);
        return;
}
```

The bubbleSort function separates the array into a sorted part and an unsorted part. After each pass, one more element is guaranteed to be in its correct position in the sorted part. In the trace, "steps" are used to separate the top part of the array still in play from the bottom part that no longer needs to be considered. We do not show here the trace of temp, pass, swapped, i and n. We start with five potential columns for passes, but we use only three of them because we quit once a pass ends without a swap. At the end of this last pass, all the elements are under the "steps":

numb

		Pass 1	Pass 2	Pass 3	Pass 4	Pass 5
[0]	9	6	2	2		
[1]	6	~~9~~ 2	~~6~~ 4	4		
[2]	2	~~9~~ 4	~~6~~ 5	5		
[3]	4	~~9~~ 5	6	6		
[4]	5	~~9~~8	8	8		
[5]	8	9	9	9		

DISCUSSION

We propose that manual tracing be an explicit, emphasized component of introductory computer science courses. This approach has been of value to us and our students and should prove to be so as well for other instructors and their students. Students trained at the outset to actively carry out manual memory traces of programs, rather than relying on debuggers or print statements, form a better connection with the inner workings of their code. This helps them overcome initial technical and emotional barriers to the challenging demands of generating correct code and it leaves them better prepared to master increasingly complex algorithms. They develop a rigorous attention to detail that helps them avoid errors in the first place and they more quickly find errors that do exist in their programs. At the end of the semester, students themselves comment that although they initially found the tracing requirement a nuisance, they came to see how the technique proved useful.

CONCLUSION

Aside from the pedagogic benefits of memory tracing, facility with this technique will benefit students when they eventually enter the workforce. Entry-level programming positions often involve modifying and maintaining code written by others. Tracing helps the programmers understand code, even if they were not involved in its original development.

REFERENCES

1. Ahmadzadeh, M., D. Elliman and C. Higgins, 2005. An analysis of patterns of debugging among novice computer science students. Proceedings of the 10th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education, June 27-29, ACM Press, Caparica, Portugal, pp: 84-88. http://portal.acm.org/citation.cfm?id=1067472
2. Bennedsen, J. and M.E. Caspersen, 2007. Failure rates in introductory programming. ACM. SIGCSE. Bull., 39: 32-36. http://portal.acm.org/citation.cfm?id=1272879
3. Johnson, E., 2006. Debugging 101. Hacknot: Essays on software development. http://appsapps.com/ebooks/?p=38
4. Lister, R., E.S. Adams, S. Fitzgerald, W. Fone, J. Hamer and M. Lindholm *et al*., 2004. A multi-national study of reading and tracing skills in novice programmers. Proceedings of the Working Group Reports from ITiCSE on Innovation and Technology in Computer Science Education, June 28-30, ACM Press, New York, pp: 119-150. http://portal.acm.org/citation.cfm?id=1041673
5. McCartney, R., J.E. Moström, K. Sanders and O. Seppala, 2004. Questions, annotations and institutions: Observations from a study of novice programmers. Proceeding of the 4th Finnish/Baltic Sea Conference on Computer Science Education, Oct. 1-3, Koli, Finland, pp: 11-19. http://www.cs.hut.fi/u/archie/KOLI/KOLI-2004-pre.pdf#page=9
6. McCracken, M., V. Almstrum, D. Diaz, M. Guzdial, D. Hagen and Y. Kolikant *et al*., 2001. A multi-national, multi-institutional study of assessment of programming skills of first-year CS students. ACM. SIGCSE. Bull., 33: 125-140. http://portal.acm.org/citation.cfm?id=572133.572137
7. Mills, H.D., 1986. Structured programming: Retrospect and prospect. IEEE. Software, 3: 58-66. DOI: 10.1109/MS.1986.229478
8. Perkins, D., C. Hancock, R. Hobbs, F. Martin and R. Simmons, 1989. Conditions of Learning in Novice Programmers. In: Studying the Novice Programmer, Soloway, E. and J. Spohrer (Eds.). Lawrence Erlbaum Associates, Hillsdale, New Jersey, ISBN: 0805800026, pp: 261-279.
9. Soloway, E., 1986. Learning to program = Learning to construct mechanisms and explanations. Commun. ACM., 29: 850-858. http://portal.acm.org/citation.cfm?id=6594
10. Vainio, V. and J. Sajaniemi, 2007. Factors in novice programmers' poor tracing skills. ACM. SIGCSE. Bull., 39: 236-240. http://portal.acm.org/citation.cfm?id=1268853