Tracing the Execution of Computer Programs – Report on a Classroom Method

Tom M. Warms **Renee Drobish**
Department of Computer Science and Engineering
Pennsylvania State University Abington College

Abstract: Part of learning how to develop computer programs is learning how to analyze programs—examples presented by the instructor and programs written by the student him or herself. One way to analyze the execution of C++ programs is by means of tracing. The tracing method is a tool with which the instructor can explain new features of the language and new programming techniques. It is also a tool that allows students to test their programs and analyze how they work and what effect possible modifications will have. The method seems to help students at both ends of the proficiency continuum, as well as some of those in the middle. Preliminary results from an ongoing classroom study on the method suggest that it is indeed effective.

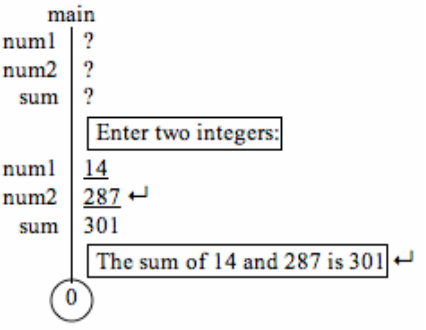
Introduction

A trace of a program or a program segment is a record of the effect of each of its executable statements. Students can benefit from tracing programs or parts of programs because it helps them learn new features of the programming language, and because it is a tool in understanding how programs work that they or others have written.

Tracing a simple program

Here, for example, is a simple program and its trace, assuming arbitrary input values of 14 and 287. (Some features of the method were described earlier.[1, 2]) The method maintains names of identifiers on the left side of a vertical line and the identifiers' values on the right. The name of the function being executed appears above the vertical line. Boxes indicate output, underlines indicate input, ↵ indicates the RETURN character, and returned values are encircled. Indeterminate values are indicated by ?. In tracing each statement, the values that resulted from tracing previous statements are available. By the time the statement to print the result is executed, the trace shows that num1, num2, and sum have values of 14, 287, and 301, respectively, and so it is these values that are printed.

```
// This program prompts the user for two integers, and calculates
// and prints their sum
#include <iostream>
using namespace std;
int main()
{
      int num1, num2, sum;
      // Prompt for input
      cout << "Enter two integers:";
      cin >> num1 >> num2;
      // Calculate sum
      sum = num1 + num2;
      // Print result
      cout << "The sum of " << num1 << " and " << num2
            << " is " << sum << endl;
      return 0;
}
```



The contents of the output screen can be inferred from the trace of a program, particularly the material that is in boxes or underlined, as well as the RETURN characters. For the program above, the output screen's contents would be as follows:

```
Enter two integers:14 287
The sum of 14 and 287 is 301
```

A good time to introduce tracing to a class is the point at which the students understand that the program statements are executed in sequence. Traces emphasize the difference between calculating a value and displaying it, and provide a means for students to go over a program step by step and predict its behavior.

It is often useful to provide values that identifiers are assumed to have, in order to trace the effects of program segments. Those identifiers and their values are listed above a horizontal line; everything below the line results from executing the segment. In fact, an effective way to display exercises is for the instructor to supply the segment and the assumed values, and to let the student write in the result of tracing the segment:

```
sum += ++term;
```

sum	12
term	-3

```
people--;
cards %= people;
```

cards	52
people	6

Tracing decision structures

To trace an if or if-else statement, write the encircled word "if" to the left of the vertical line, the Boolean value of the condition across the line, and on succeeding lines the result of executing the if-true statement or the if-false statement. The material in italics annotates the trace, but is not part of the trace itself. In tracing switch statements, the value of the selector expression is again written across the vertical line, followed by the trace of the selected clause:

```
if (item > large)
      large = item;
```

(if)

item	15
large	7
true	*item > large*
large	15

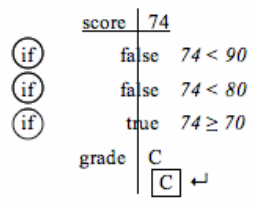
```
switch (party)
{
      case 'D': numDs++; break;
      case 'R': numRs++; break;
      default: numIs++; break;
}
```

(switch)

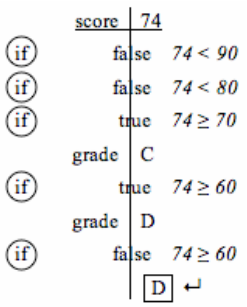
numDs	102
numRs	55
numIs	32
party	D
D	
numDs	103

Tracing is a useful way of getting across the difference between nested if-elses and sequential ifs. From left to right below are a nested if-else; its trace assuming score is 74; sequential ifs; a trace, again assuming score is 74:

```
if (score >= 90)
    grade = 'A';
else if (score >= 80)
    grade = 'B';
else if (score >= 70)
    grade = 'C';
else if (score >= 60)
    grade = 'D';
else
    grade = 'F';
cout << grade << endl;
```



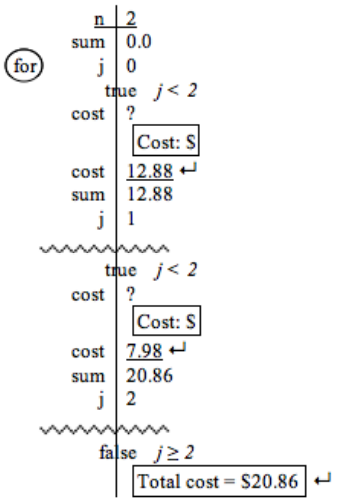

```
if (score >= 90)
    grade = 'A';
if (score >= 80)
    grade = 'B';
if (score >= 70)
    grade = 'C';
if (score >= 60)
    grade = 'D';
if (score < 60
    grade = 'F';
cout << grade << endl;
```



Tracing looping structures

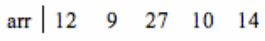
In the trace of a looping structure, the name of the structure is encircled and written to the left of the line. In the for loop, a squiggly line is drawn across the vertical line after each execution of the update clause. Here is a segment containing a for loop, and its trace, with n assumed to be 2 and the input values assumed to be 12.88 and 7.98:

```
double sum = 0.0;
for (int j = 0; j < n; j++)
[
      double cost;
      cout << "Cost: $";
      cin >> cost;
      sum += cost;
}
cout << "Total cost = $" << fixed
      << showpoint << setprecision(2)
      << sum << endl;
```

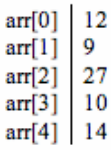


Tracing array operations

The values of the elements of an array can be represented linearly, as in

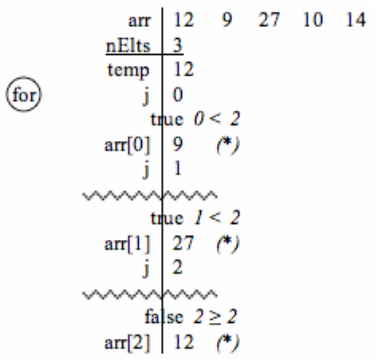


which represents an array of 5 elements: arr[0] is 12, arr[1] is 9, and so on. An alternative representation is to list each element separately:



In the following segment, the first three elements of array arr are rotated forward one place, with arr[0] replacing arr[2]. The last two elements of the array do not participate in the rotation.

```
int temp = arr[0];
for (int j = 0; j < nElts - 1; j++)
      arr[j] = arr[j + 1];
arr[nElts – 1] = temp;
```

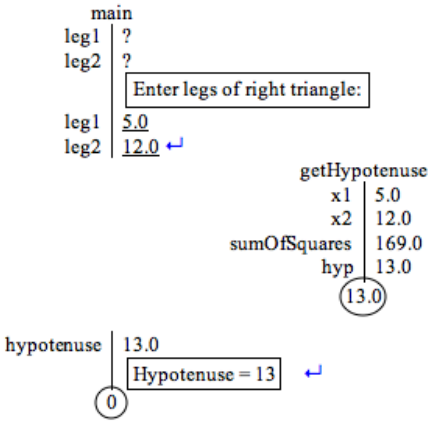


After the segment has been executed, values for arr[0], arr[1], and arr[2] can be read from the starred lines, which provide their most recent values, while the most recent values for arr[3] and arr[4] can still be read from the first line of the trace.

Tracing value-returning functions

In tracing a value-returning function, the trace moves to the right, and the name of the function is written above a continuation of the vertical line. The first values to be entered at that point are those of the formal parameters. Below is a program that uses a function to calculate the hypotenuse of a right triangle, along with a trace that assumes input values for the legs to be 5.0 and 12.0. When the main program is executed, leg1 and leg2 are in scope. As the function is executed, the variables that are in scope are x1, x2, sumOfSquares, and hyp. Execution of the function terminates with a returned value of 13.0, which is encircled. The trace returns to the original vertical line, and leg1 and leg2 are in scope again.

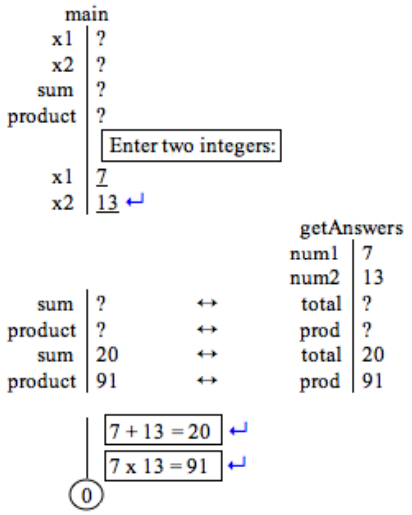
```
// Program to prompt the user for the legs of a right
// triangle and calculate and print the hypotenuse
#include <iostream>
#include <cmath>
using namespace std;
double getHypotenuse (double leg1, double leg2);
int main()
{
      double leg1, leg2;

      // Prompt for legs of triangle
      cout << "Enter legs of right triangle:";
      cin >> leg1 >> leg2;

      // Calculate hypotenuse
      double hypotenuse = getHypotenuse(leg1, leg2);

      // Print hypotenuse
      cout << "Hypotenuse = "  << hypotenuse << endl;
      return 0;
}

double getHypotenuse (double x1, double x2)
// getHypotenuse calculates the hypotenuse of a
// right triangle whose legs are x1 and x2
{
      double sumOfSquares = pow(x1, 2.0) + pow(x2, 2.0);
      double hyp = sqrt(sumOfSquares);
      return hyp;
}
```



Tracing functions with reference parameters

When a formal parameter is a reference parameter, any change in its value is reflected in a change in the value of the corresponding actual parameter. The tracing system uses a double arrow to indicate this relationship between actual and reference parameter. In the following program, a function named getAnswers calculates the sum and product of two input numbers. When control is transferred to getAnswers, value parameters num1 and num2 take on the values of the corresponding actual parameters x1 and x2. Reference parameters total and prod also take on the values of the corresponding actual parameters, sum and product, respectively, which are indeterminate at this point. Then the function body of getAnswers is executed. These statements result in the calculation of new values for total and prod, and therefore for sum and product as well. When control returns to the main program, x1 and x2 have their original values, but sum and product already have new values. The trace assumes that the input values are 7 and 13.

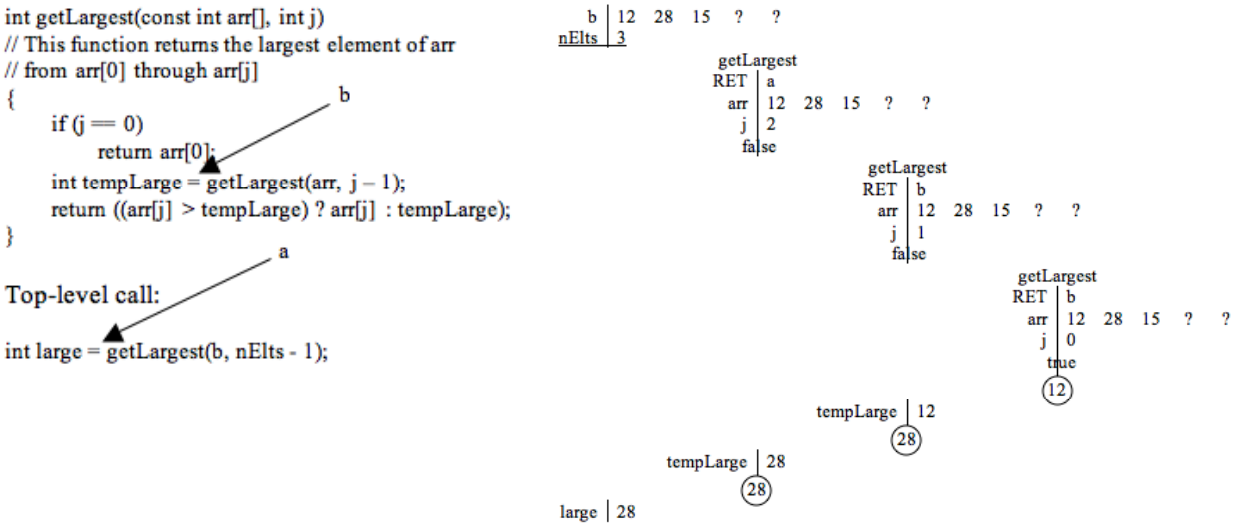
```
// Program to prompt user for two integers and calculate
// and print their sum and product.
#include <iostream>
using namespace std;
void getAnswers (int num1, int num2, int &total, int &prod);
int main()
{
      int x1, x2, sum, product;

      // Prompt for input
      cout << "Enter two integers:";
      cin >> x1 >> x2;

      // Call function to do the calculations
      getAnswers(x1, x2, sum, product);

      // Print results
      cout <<  x1 << " + " << x2 << " = " << sum < < endl;
      cout <<  x1 << " x " << x2 << " = " << product < < endl;
      return 0;
}
```



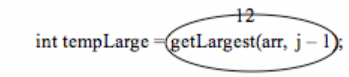

```
void getAnswers (int num1, int num2, int &total, int &prod)
// Function getAnswers calculates the sum and product of
// its first two parameters and places the answers
// in the third and fourth parameters
{
      total = num1 + num2;
      prod = num1 * num2;
}
```

Tracing recursive functions

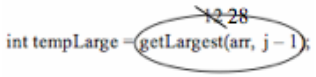
The only new element that is introduced by recursion is the return address. Each time a recursive function is called, the trace moves to the right and the name of the function is placed above a continuation of the vertical line. The return address is entered as the value of RET and the trace continues as with any function. In the following example, a recursive function, getLargest, is used to calculate the largest element of an array. There are two return addresses: one, which is marked a, is at the point where the result of the calculation is assigned to large in the top-level call, and the other, marked b, is where the result of the recursive call is assigned to tempLarge. The top-level call as well as the function are shown on the left. The trace on the right shows that the first 3 array elements have the assumed values 12, 28, and 15, and nElts is assumed to equal 3.



The trace shows that there are three calls to getLargest: the first is the top-level call and the two others are recursive calls. When control returns from the third call with the indicated value of 12, that value is assigned to tempLarge in the second call as a result of the statement



The next statement compares this value with arr[1], which is 28, and returns the larger value, 28, to point b in the first call:



In the first call, this value is compared with arr[2], which is 15, and so 28 is returned to point a in the top-level call, where it is assigned to large.

Tracing programs with dynamic variables

The tracing tool is also useful in helping students understand how dynamic data structures such as linked lists and binary search trees work. Recursive insertion of an element into a binary search tree, in which each node contains a string as well as pointers to left and right subtrees, is a typical example. The definition of the node as well as the insertion routine and the top-level call segment, are given below, along with a trace of the insertion of a new word into a tree in which root is NULL. Return address a is in the top-level segment, and b is just before the end of insert's function body.

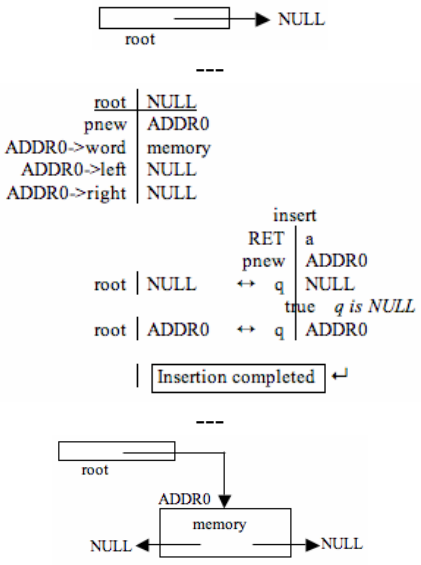
```
struct node
{
      string word;
      node *left, *right;
};

void insert(node *pnew, node *&q)
// This function inserts the node pointed to by pnew
// into the binary search tree pointed to by q
{
      if (q == NULL)
            q = pnew;              // (1)
      else if (pnew->word  < q->word)
            insert(pnew, q->left);
      else
            insert(pnew, q->right);
}
```

b

Top-level segment:

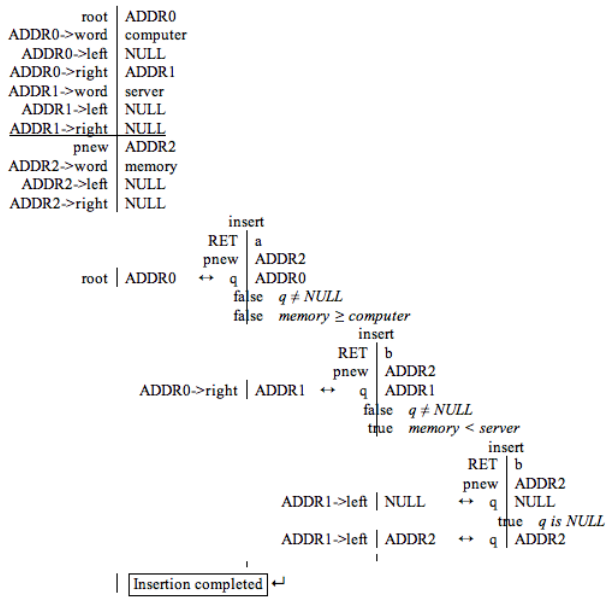
a

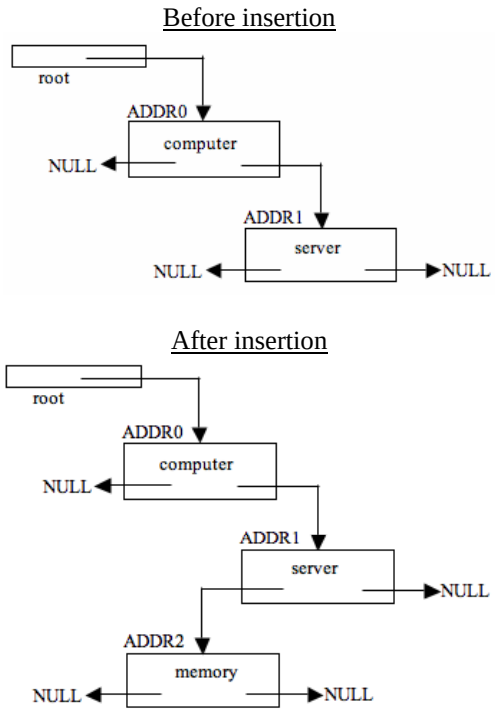
```
node *pnew = new node;
pnew->word  = "memory";
pnew->left  = NULL;
pnew->right = NULL;
insert(pnew, root);
cout << "Insertion completed\n";
```



When the top-level statements are executed, the system allocates memory for a node and its contents are initialized to the indicated values; the trace uses the symbol ADDR0 to represent the new node's address. Thus the value of ADDR0->word is "memory," and the left and right pointers at that location are both NULL. In addition, ADDR0 is assigned to pnew. When insert is called, root is the actual parameter corresponding to reference parameter q. The statement labeled (1) causes both to be set to ADDR0, so that root now points to this new node. The "before" and "after" representations of the tree are, respectively, above and below the trace.

A second trace shows how the insertion routine can insert an element into its alphabetical position in a non-empty tree. In the following, the root word is "computer"; its right child is "server". Again, the word "memory" is to be inserted. This time, when the actual insertion is performed the actual parameter corresponding to q is ADDR1->left, making "memory" a left child of "server." "Before" and "after" representations of the tree are given below, as well as the trace.





Other features of C++

There are a number of features that have not been touched on in this description of tracing. Global identifiers are an example; they can be handled in the system, but awkwardly. Tracing of objects and other features of C++, as well as an analysis of the semantics of tracing, will be described elsewhere.

Using tracing in the classroom

The tracing system is meant to clarify various features of the C++ language for students. It has over the years been quite satisfactory in this role: students frequently ask "How do you trace that?" when one of the authors introduces a new feature of the language. It therefore provides a way to communicate with students—a language that describes how the computer carries out instructions. It is also a useful vehicle for the instructor to display the behavior of algorithms and for students to test algorithms that they have written.

Experience has shown that it is best to wait until students have caught on to the idea of sequential execution of programs before introducing tracing. A student who is having trouble with the basic concepts of programming will only get more confused if confronted with a new set of notations. But if an instructor introduces tracing, it is worthwhile to give students the opportunity to learn it well. Many students who learn the system truly enjoy using it. And tracing seems to help students understand subprograms, recursion, and arrays, as well as many algorithms. It is the authors' experience that many students who are doing well in an introductory course "hit a wall" at arrays; tracing helps surmount the wall. Additionally, there is no other method known to the authors that is as useful as tracing in explaining the execution of routines on recursive data structures.

The system is meant to be used for short programs. Students who are in doubt about how an algorithm works should be encouraged to "cut the program down to size" before tracing it. When this is done, the system is used to best advantage.

It's always important to remember that tracing can be part of the process of program development; it is not put forward as a substitute for actually writing programs.

Questions

The authors' subjective experience is that tracing is indeed a useful method. However, studies are needed to test whether this is objectively so. There is also a feeling that watching demonstrations of tracing, rather than actually doing tracing is enough for many students; a study would test whether this is so. Also unknown is whether and to what extent understanding of how recursive programs are executed helps students write the programs.

A Preliminary Study

A preliminary study was recently carried out in the classroom, in which tracing was compared with a typical tabular system of hand-checking. One group was taught the tracing method decribed in this paper, and a control group was taught a tabular system to hand-check loops and if/else statements. Although the two groups performed comparably on quizzes and examinations during most of the semester, there are indications that the group that was taught the tracing method benefited from this in the work that they did toward the end of the course, particularly on the final examination. Also, students in the tracing group who did well in the final examination seemed to feel that the tracing method did indeed help them write programs.

Conclusions

The tracing system, as presented here, has been used in the classroom for a number of years. A preliminary study provides indications that students do benefit from using the system. Additional testing in the classroom is in the planning stages.

Acknowledgment

The authors wish to thank Renata Engel of the Penn State College of Engineering for her encouragement, Sarah Zappe of the Schreyer Institute for Teaching Excellence at Penn State for her assistance, and the students in Computer Science 101 Sections 1 and 2 during Fall 2006 at Penn State Abington for their participation in the preliminary study.

Bibliography

[1] Warms, Tom M., "Tracing the Execution of C++ Programs," *Inroads - the SIGCSE Bulletin*, Vol. 33, No. 4, (2001), 64-67.

[2] Warms, Tom M., " The Power of Notation: Modeling Pointer Operations," *Inroads - the SIGCSE Bulletin*, Vol. 37, No. 2, (2005), 41-45.