Tracing & Trace Tables

Topics:

+ Trace Table Header
+ Populating a Trace Table
+ Trace Table List

Resources:

- `prog1.cpp`
- `prog2.cpp`
- `prog3.cpp`

Introduction

Tracing, which is the process of determining what a program is doing, is one of the most essential abilities every programmer needs to know how to do. To assist you with tracing, you can use a *trace table*. It is a table that keeps track of the changes to important components in the program for a partical instance of the program (a call). Last, it is important to note that a trace table should be constructed for individual functions.

Trace Table Header

When you are constructing a trace table, you start with the header row. It consists of components from the program that are the *mutable* (changeable) structures such as variables, arrays, conditions of control structures and so on. Furthermore, it includes an output column whenever the program is displaying something (contains cout statements), a return column whenever the program (function) is returning a value [this is be discussed more when we cover functions in more details], and the optional step column whenever you need to keep track of your steps. To practice, let us determine the header for the code below.

```
int main()
{
 int x, y = 4;
 cin >> x;
 cout << "result:  "<< x + y;
 return 0;
}
```

If your header consists of the columns *step*, *x*, *y*, *output* and *return*, you are correct. From the above code, you can see that there are two variables `x` and `y`; hence, there should be a column for each of them in the trace table header. Next, the code is displaying a message (using a cout statement) and contains a return statement, which means there should be a output column and return column as well. Notice that we do not include an input column. The reason for this is because inputs are stored directly into variables. Hence, whenever you come across an cin statement, you are just modifying the value of its variable argument.

Sometimes identifiers and/or conditions of control structures are long. Instead of the writing them in the header, you can use substitutions and a *legend*, which is a list that explains what substituted symbols stand for. For instance, consider the following code and trace table header.

```
int main()
{
 int numerator, denominator, remainder, whole;

 cout << "Enter numerator: ";
 cin >> numerator;
 cout << "Enter denominator: ";
 cin >> denominator;

 whole = numerator / denominator;
 remainder = numerator % denominator;
 cout << whole << " + "<< remainder << "/"<< denominator << '\n';
 return 0;
}
```

D = `denominator`
N = `numerator`
R = `remainder`
W = `whole`

step	N	D	W	R	output	return

The above header would be no different from a header with the names of the variables written out except using it saves space. Thus, the advantage (or need) of using a legend is to avoid the header exceeding a single line.

Construct the header of the following exercises.

Exercise 1.

```
int main()
{
 double real, imaginary;

 cout << "Enter real component: ";
 cin >> real;
 cout << "Enter imaginary component: ";
 cin >> imaginary;

 cout << real << " + "<< imaginary << "i\n";
 return 0;
}
```

Exercise 2.

```
int main()
{
 cout << "My name is John Doe\n";
 cout << "Hello World\n";
 return 0;
}
```

Exercise 3.

```
int main()
{
 int n = 7;
 n = n * n * n * n;
 cout << n << "\n";
 return 0;
}
```

Populating a Trace Table

After you construct the header of a trace table, you are ready to populate it. Whenever you are populating a trace table, there are a few things you must know. First, programs are sequential. Hence, you must begin at the beginning of the program, and either, work your way to the end of the body of the program or to an executable return statement. It is important to note that once a return statement is executed, the program terminates regardless of any code that follows it. Therefore, if your trace table have a return column and you populate it, your trace is over; otherwise, you continue until you make it to the end of the body of the program.

Second, only one thing changes during a single step unless either the header of the program has a list of parameters, which means, the first step will list all the values of the parameters; or, the program is executing a function call that takes reference parameters. Besides those cases, you will only be populating the step column and one other column for every row even if multiple variables are initialized on a single line. Remember initialization occurs from left to right. Furthermore, the step column does not correspond with the lines in the code; but rather, the changes in the code. Therefore, some lines do not contribute to the trace table; whereas, other lines contribute multiple times to the trace table.

Third, some column such as the output column, will require you to delegate its value to a separate table. Whenever you are populating these columns, you will place an identification tag, which would be the name of the separate table, in the column section; and then, you add the value to the separate table. We use separate tables for these columns because adding the content directly to the column will make the trace table harder to construct and redundant.

Last, whenever a program receives inputs, a list of inputs will be provided to perform the trace. Each input in the list corresponds to the cin statement in the same order; that is, the first input is for the first cin statement, the second input for the second cin statement and so on.

Let us generate the trace table for the code below

```
int main()
{
 int x = 6, y = 4, r;
 r = x * x - y * y;
 cout << "result:  "<< r << "\n";
 return 0;
}
```

The trace table should look something like this

step	x	y	r	output	return
1	6				
2		4			
3			20		
4				out	
5					0

out
result: 20

Notice that the values of populated columns are not repeated in future steps. This is because it is understood that the value of the column remains the same until it is reassigned. Hence, to generate the value of the `r` column, which is dependent on the values of the `x` and `y` columns, we only needed to look at the last changes to the `x` and `y` columns.

Let us work with another example that takes inputs

```
int main()
{
 int sum, i;

 cout << "Enter an integer: ";
 cin >> i;

 sum = i + (i + 1) + (i + 2);
 cout << sum << '\n';
 return 0;
}
```

Now, the trace table given the input (`9`) is

step	i	sum	output	return
1			out	
2	9			
3		30		
4			out	
5				0

```
      out
Enter an integer:
30
```

To land the procedure, let us go through a thorough construction of a trace table for the code below that uses the input `(45)`

```
01  int main()
02  {
03   bool m2, m3, m6;
04   int n;
05
06   cout << "Enter a positive number:  ";
07   cin >> n;
08   cout << boolalpha << "It is ";
09
10   m2 = n % 2 == 0;
11
12   cout << m2 << "that"<< n << "is divisible by 2 and it is ";
13
14   m3 = n % 3 == 0;
15
16   cout << m3 << "that"<< n << "is divisible by 3.\nTherefore, it is";
17
18   m6 = m2 && m3;
19
20   cout << m6 << "that"<< n << "is divisible by 6.\n";
21   return 0;
22  }
```

First, after analyzing the code, we see that there are variable declarations on lines 3 and 4; Furthermore, there are cout statements on lines 6, 8, 12, 1 and 20, and there is a return statement on line 21. Hence, our trace table header should be

step	***m2***	***m3***	***m6***	***n***	***output***	***return***

Now, we are ready to populate the table. On line 1, the function header does not contain any parameters, so we will skip this line. Likewise, we will skip the lines 3 and 4 since they are just variable declaration. However, on line 6 we come across our first cout statement. Since it produces a display, we will add it to our trace table as step 1.

step	***m2***	***m3***	***m6***	***n***	***output***	***return***
1					out	

We place *out* underneath the ***output*** column, and place the value of the cout statment in the out table as follows

out
`Enter a positive number:`

The next line in the code is a cin statement. It is storing the input in the variable *n*, thus the second step in the trace table will be 45 (the input) assigned to ***n***

step	***m2***	***m3***	***m6***	***n***	***output***	***return***
1					out	
2				45		

Now, we are up to line 8 in the code. It is another cout statement. Its arguments are the `boolalpha()` function [special calling procedure than normal function calls] and a string literal. The `boolalpha()` function formats the display to have boolean arguements display as the string literals `"true"` and `"false"` rather than 1 and 0 respectively, but the function does not display anything. So step 3 is an addition to the ***output*** column and out table

step	***m2***	***m3***	***m6***	***n***	***output***	***return***
1					out	
2				45		
3					out	

out
`Enter a positive number:`
`It is`

Next, line 10 is performing a variable assignment to *m2* which means it needs to be added to the table as our next step.

step	***m2***	***m3***	***m6***	***n***	***output***	***return***
1					out	
2				45		
3					out	
4	false					

Afterwards, we have a cout statement on line 12 which will be the 5th step in the table

step	***m2***	***m3***	***m6***	***n***	***output***	***return***
1					out	
2				45		
3					out	
4	false					
5					out	

out
`Enter a positive number:`
`It is false that 45 is divisible by 2 and it is`

Notice that the new content is not written on a new line. This is because a newline character has not been read. The only other time you can put a new line in the display is if the cout statement comes after a cin statement.

Now, the next line in the code that contributes to the table is line 14. It assigns true, which is the result of the expression, to *m3*.

step	***m2***	***m3***	***m6***	***n***	***output***	***return***
1					out	
2				45		
3					out	
4	false					
5					out	
6		true				

Once again, we have a cout statement on line 16 that will be the next step in the table.

step	***m2***	***m3***	***m6***	***n***	***output***	***return***
1					out	
2				45		
3					out	
4	false					
5					out	
6		true				
7					out	

out

```
Enter a positive number:
It is false that 45 is divisible by 2 and it is true that 45 is divisible by 3.
Therefore, it is
```

The next step in the table is the final assignment on line 18, which assigns false to *m6*.

step	***m2***	***m3***	***m6***	***n***	***output***	***return***
1					out	
2				45		
3					out	
4	false					
5					out	
6		true				
7					out	
8			false			

Afterwards, the final cout statement is the next step in the table

step	*m2*	*m3*	*m6*	*n*	*output*	*return*
1					out	
2				45		
3					out	
4	false					
5					out	
6		true				
7					out	
8			false			
9					out	

out

```
Enter a positive number:
It is false that 45 is divisible by 2 and it is true that 45 is divisible by 3.
Therefore, it is false that 45 is divisible by 6.
```

Finally, the last step in the table is generated from the return statement. Hence, the trace table in its entirety is

step	*m2*	*m3*	*m6*	*n*	*output*	*return*
1					out	
2				45		
3					out	
4	false					
5					out	
6		true				
7					out	
8			false			
9					out	
10						0

out

```
Enter a positive number:
It is false that 45 is divisible by 2 and it is true that 45 is divisible by 3.
Therefore, it is false that 45 is divisible by 6.
```

Now, complete the following exercises.

Exercise 4. Generate the trace table for Exercise 1 given the inputs (`-6`, `12`).

Exercise 5. Generate the trace table for Exercise 2.

Exercise 6. Generate the trace table for Exercise 3.

Trace Table List

To construct a trace table using a plain text editor is hard; they are ideal for spreadsheets. However, if using a spreadsheet is not an option, you can write the trace table as a list. Each row of the list follows the following format

$$\textit{step-number}.\ \textit{column-name} := \textit{column-value}$$

For instance, the last trace table we constructed as a list would be

1. `output := out`
2. `n := 45`
3. `output := out`
4. `m2 := false`
5. `output := out`
6. `m3 := true`
7. `output := out`
8. `m6 := false`
9. `output := out`
10. `return := 0`

out

`Enter a positive number:`
`It is false that 45 is divisible by 2 and it is true that 45 is divisible by 3.`
`Therefore, it is false that 45 is divisible by 6.`

Finally, if a step consists of multiple values, you just need to use the comma as follows

$$\textit{step-number}.\ \textit{column-name} := \textit{column-value}\big[, \textit{column-name} := \textit{column-value}\big]^*$$