ICS3U: Trace Tables

A trace table is an organized chart that allows a programmer to track changes in the values of variables. Trace tables are generally used for debugging code to fix logical errors. As programs become more complex and use a greater number of variables, it is useful to be able to determine their values as a program executes.

Each line in a trace table is used to record a change in one variable. There must be enough columns to account for all variables that are being monitored. If a variable has not been declared or initialized, it should *not* have a value in the trace table. Do *not* use zero, as this is an actual value, and is different from "has no value" or "does not exist".

The code below computes the average of an arbitrary number of positive integers.

```
num_values = 0
running_total = 0
value = int(input("Enter a positive integer (zero or less to stop): "))
while value > 0:
   running_total += value
   num_values += 1
   value = int(input("Enter another (zero or less to stop): "))
if num_values > 0:
   average = running_total / num_values
   print("The average is", average)
else:
   print("No values were entered.")
```

The trace table for this program would track four variables – `num_values`, `running_total`, `value` and `average` – as they changed during program execution. Let's examine the case where the user enters the positive values 3, 5 and 10, followed by 0 to stop.

num_values	running_total	value	average
0	--	--	--
0	0	--	--
0	0	3	--
0	3	3	--
1	3	3	--
1	3	5	--
1	8	5	--
2	8	5	--
2	8	10	--
2	18	10	--
3	18	10	--
3	18	0	--
3	18	0	6

To make things easier to follow, each iteration through the loop is marked by shading in the table. The first three lines of the table show the initialization of the variables `num_values` and `running_total`, as well as the entry of the first value for `value`. Since the condition `value > 0` is `True`, the loop begins its first iteration. The next three lines (shaded) correspond to this iteration, in which both `num_values` and `running_total` are incremented. A new `value`, 5, is read and the loop begins its second iteration. This is followed by a third iteration of the loop for the `value` of 10. At the end of this iteration, the user enters a `value` of 0, thus ending the loop. The final line of the table shows the calculation for `average`.

Trace tables can get large relatively quickly if there are a lot of variables or changes. Making only one change per line can require tens or even hundreds of lines. As such, they are often compressed, with each line representing a single iteration of a loop. This really only works well if each variable is changed at most once in a loop – multiple changes to a variable would require the same variable to appear in several columns, in the order it is changed.

Consider the code below, which attempts to calculate the factorial, $n!$, of a user-entered positive integer. Recall that $n!=n\times(n-1)\times(n-2)\times\ldots\times2\times1$. For example, $4!=4\times3\times2\times1=24$; however, the code produces a value of 64 when $n=4$.

```
num = int(input("Enter a positive integer: "))
fact = 1
for count in range(num, 1, -1):
    fact *= num
print(fact)
```

A trace table for this program will require three variables: `count`, `num`, and `fact`. A condensed trace table is below. Each line represents one iteration of the loop.

count	num	fact
--	4	1
4	4	4
3	4	16
2	4	64

A pattern can be seen in the `fact` column: the value quadruples for each iteration of the loop. This illustrates a problem with the code. The value of fact should multiply by a decreasing amount each time. Examining the code more closely, the wrong variable has been used in the multiplication. Instead of `num`, which has a fixed value of 4, fact should be multiplied by `count` instead. The corrected code is below.

```
num = int(input("Enter a positive integer: "))
fact = 1
for count in range(num, 1, -1):
    fact *= count
print(fact)
```

Instead of generating a trace table, programmers often use a **debugger** to track the values of variables in their programs. If a debugger is a component of your IDE of choice, you may wish to investigate how to use it, so that you do not need to track changes manually.