Tracing and Visualizing C Programs Using C Coding Tutor: Learning Activities for CCPROG1

Preliminary Task: please click this link for Check-up Quiz #2 and answer a simple non-graded check-up quiz on functions and parameter passing. The form is only accessible to a DLSU domain; please log-in to your DLSU account in case you cannot access it the first time around. The quiz assumes that you have already covered the topics on functions in your CCPROG1 class. The objective of the quiz is to determine possible misconceptions which we intend to clarify and remediate as part of the learning activities. The quiz answers can be found in the footnote on page 5 of this document.

MODULE 3: Tracing and Visualizing Functions

It is assumed that you have covered the topic on functions in your CCPROG1 class. For this module, we will visualize what happens during function calls, parameter passing and returning values from a called function.

Learning Activity 3.1: Visualizing a void function without parameter passing (part 1)

We will visualize a simple program with a main function that calls a void function without passing any parameter.

Learning Outcome: The student can trace the control flow in function calls.

Instructions:

1. Encode the source code shown in Figure 3.1. Or if you want to omit the encoding step, you may simply click this link to open a browser window with a pre-encoded program.

```
1  #include <stdio.h>
2
3  // Test() is a void function without any parameter
4  void Test() {
5      printf("Inside Test() function.\n");
6  }
7
8  int main() {
9      Test();  // main() calls Test()
10     return 0;
11 }
```

Figure 3.1: A main function and a void function without any parameter

2. Let us see a visualization of the steps during execution. Press "Visualize Execution" button, and then the "Next" button. As shown in Figure 3.2, the program execution starts from the main function as it is the current active function indicated on the right half of the screen. The red arrow points to Line 9 which means that the function call to Test will be executed next.

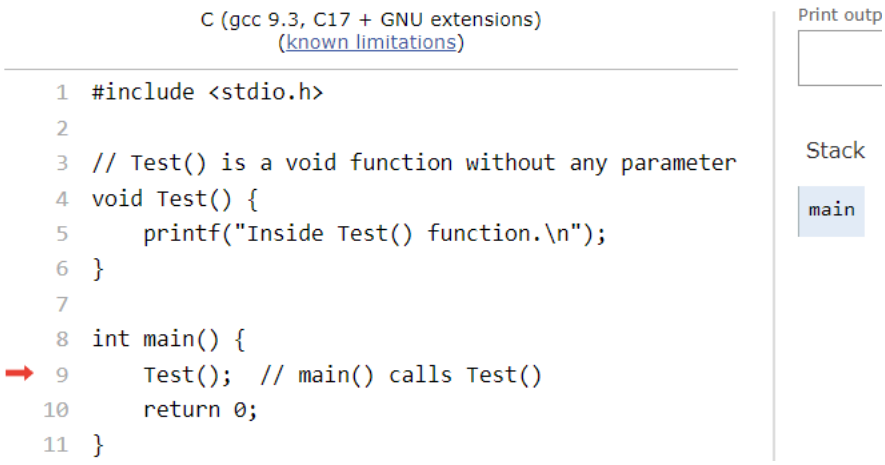

Figure 3.2: The next line to execute is the Line 9, i.e., main will call Test function.

3. Press the "Next" button again. As shown in Figure 3.3, the green arrow points to Line 9 which means that the function call to Test was executed. The red arrow points to Line 5 which indicates that the printf statement inside Test will be executed next. The right side of the screen shows a bluish gray box labeled as Test. Appearing on top of it is the box labeled main which is now in white. In C Tutor, this means that Test is the currently executing function, and that main is waiting (suspended) and can only resume its own execution only after Test has finished its task. We can also say that Test is at the top of the stack, and that main is at the bottom of the stack. *[note: the concept of stack will be covered in your future subject on "Data Structures and Algorithms"]*

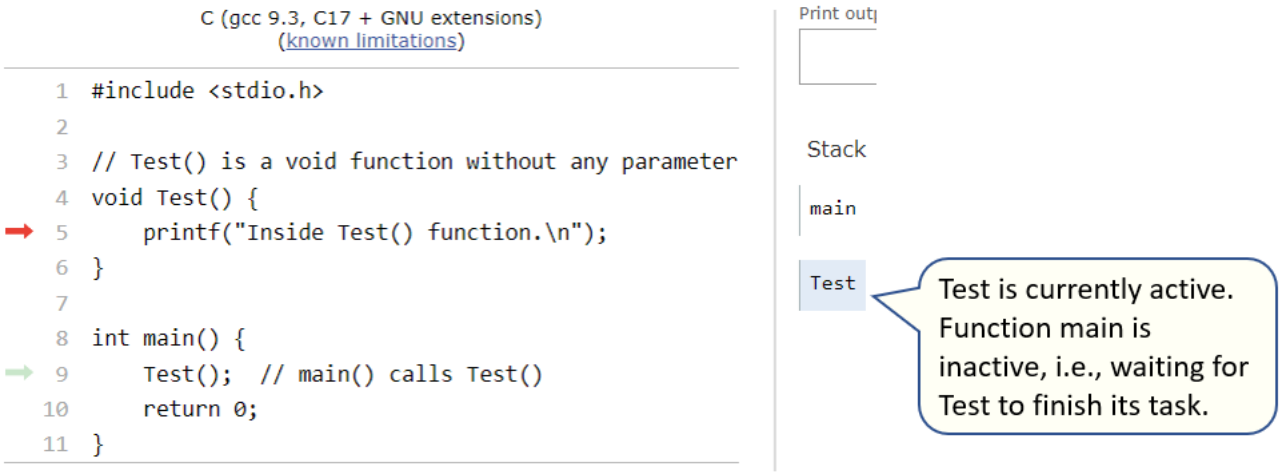

Figure 3.3: Test is the currently executing function. The next line to execute is Line 5. The right half of the screen also shows that the main function is waiting for Test to finish its task.

4. Press the “Next” button again. As shown in Figure 3.4, Line 5 has just executed, and the result of the printf statement is displayed in the “Print output” box. The red arrow points to Line 6 which means that Test function is about to finish its task.

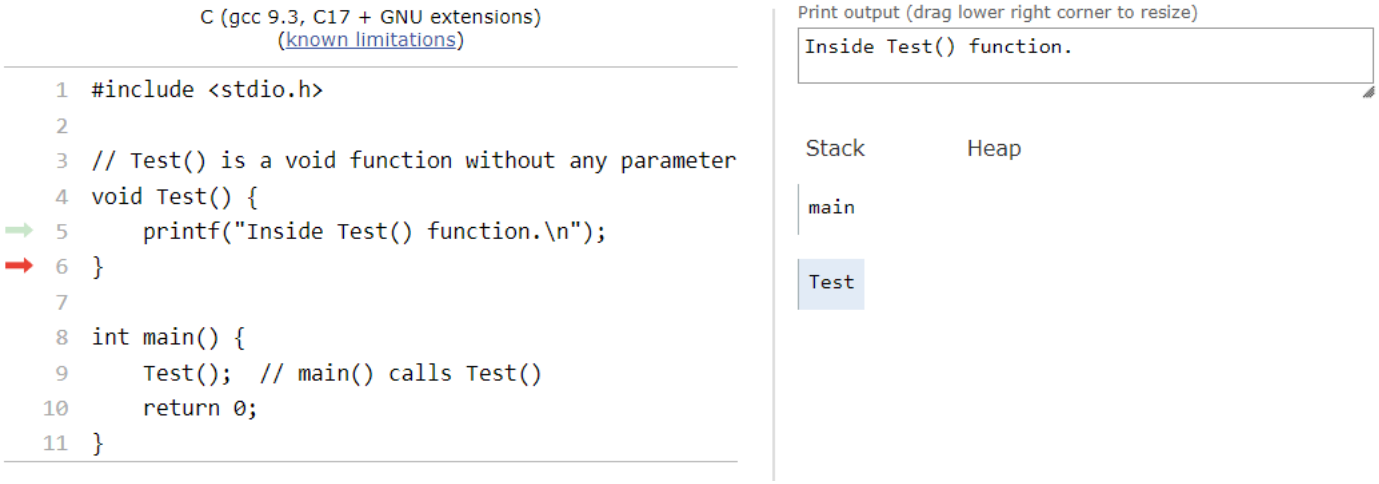

Figure 3.4: Red arrow is pointing at Line 6 indicating that Test is about to finish its task.

5. Press “Next” twice. As shown in Figure 3.5, line 6 finished executing, and that control is now back to the main function. This is also visualized on the right half of the screen where the bluish gray box shows main. Notice that since Test finished its task, it no longer appears in the Stack visualization. The next line to execute is Line 10. Press “Next” one more time to finish the program execution.

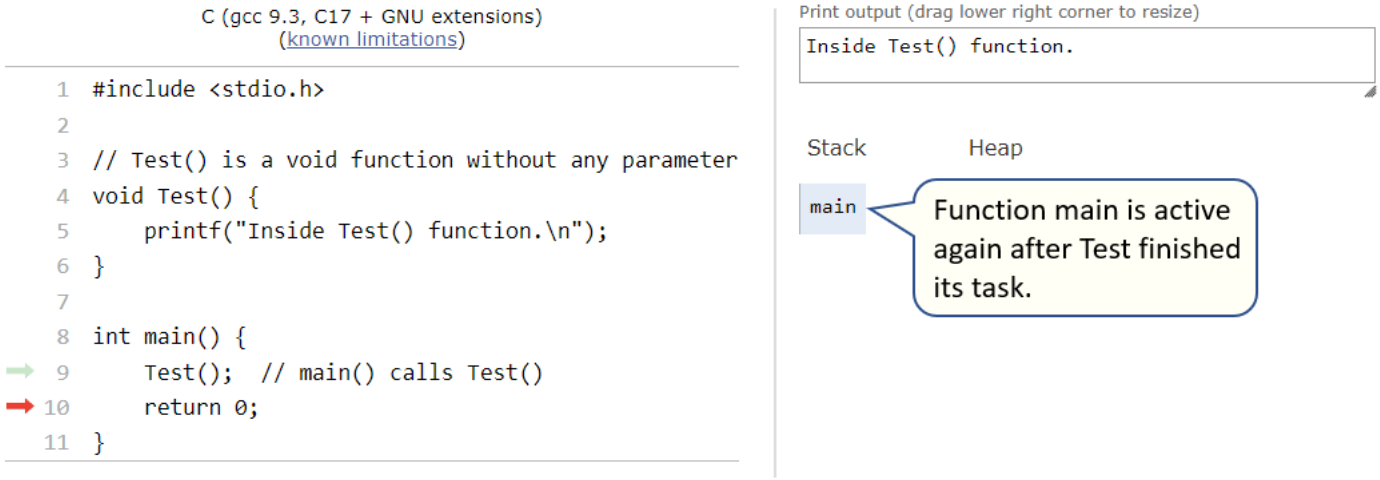

Figure 3.5: Test function has finished its task. Function main is now active again, and Line 10 is the next to execute.

Learning Activity 3.2: Visualizing a function call without parameter passing (part 2)

The previous code only has one user-defined function other than main. We have seen visually the control flow changes when a main function does a function call. The important idea that we learned is that the calling function will have to wait, i.e., become inactive until the function that it called has finished its own task. The same idea applies no matter how many user-defined functions are present in a program.

To gain more insight, explore, independently on your own, what happens when the main function calls a function which in turn calls another function.

Learning Outcome: The student can trace the control flow in function calls.

Instructions:

1. Encode the source code shown in Figure 3.6. Or if you want to omit the encoding step, you may simply click this link to open a browser window with a pre-encoded program.

```c
#include <stdio.h>

// Test() is a void function without any parameter
void Hello() {
    printf("Inside Hello().\n");
}

// AnotherTest() is a void function without any parameter
void Hola(){
    Hello();  // Hola() in turn calls Hello()
    printf("Inside Hola().\n");
}

int main() {
    Hola(); // main() calls Hola()
    return 0;
}
```

Figure 3.6: main calls Hola which in turn calls Hello

2. Trace and visualize the execution of the program by clicking on the "Next" button until the end of the program. Pay attention to what happens as you progress through each step.
3. Edit the source code and add another void function that in turn will call Hola. Feel free to add whatever statement you want in that new function. Then call that function inside main. Trace and visualize the program execution.

Learning Activity 3.3: Visualizing a function call with parameter passing

For this activity, we will explain and visualize what happens in a program that involves parameter passing. You learned in your CCPROG1 that functions pass the **value** of a parameter from the calling to the called function. You may want to refer back to your CCPROG1 notes about **actual parameter** and **formal parameter** as you go along this activity.

Learning Outcome: The student can trace a function call that involves parameter passing and returning of a value

Instructions:

1. Encode the source code shown in Figure 3.7. Or if you want to omit the encoding step, you may simply click this link to open a browser window with a pre-encoded program.

   Two functions are present, namely main and A. Function A has two formal parameters named x and y. It returns as a result the value of x – y. Function main calls A in Lines 10 to 13. The return values are stored in temp1, temp2, temp3 and temp4.

   The code is the same one given in Check-up Quiz #2. Click this link if you have not answered it. Please refer to the footnote for the answers to the quiz[1].

```
1  #include <stdio.h>
2
3  // function A() is an int function with two parameters
4  int A(int x, int y) {
5      return x - y;
6  }
7
8  int main() {
9      int x = 10, y = 5; // variable definition
10     int temp1, temp2, temp3, temp4; // variable declaration
11
12     // the following are function calls to A()
13     temp1 = A(25, 5);
14     temp2 = A(x, y);
15     temp3 = A(y, x);
16     temp4 = A(A(1, 2), A(4, 3));
17     printf("%d %d %d %d\n", temp1, temp2, temp3, temp4);
18     return 0;
19 }
```

Figure 3.7: Sample program with several function calls and parameter passing

[1] Check-up Quiz #2 Answers:
(a) 2 functions were defined, the first was A and the second was main.
(b) 6 times
(c) 20 5 -5 -2

2. Let us see a visualization of the program flow. Press “Visualize Execution” button, and then press the “Next” button three times. Observe the visualization closely as you go through the first three steps. As shown on the right half of the screenshot in Figure 3.8, main is the current active function, and it has 6 local integer variables. The values of x and y were defined as 10 and 5 respectively, while temp1 to temp4 contain garbage values visualized as question marks. The red arrow points to Line 13 which is the first time that main will call function A.

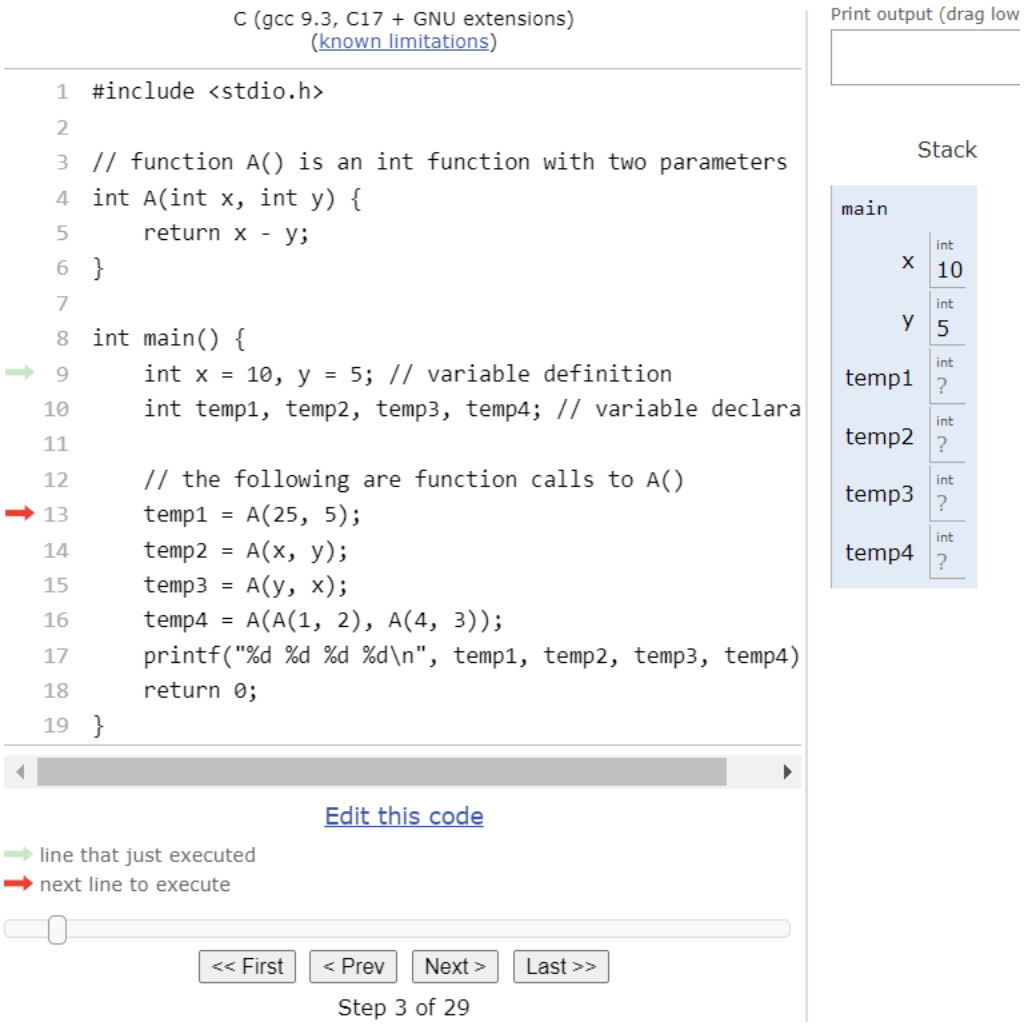

Figure 3.8: The next line to execute is Line 13, i.e., main will call function A for the first time.

3. Press the "Next" button again. We're now in Step 4. As shown in Figure 3.9, notice that the green arrow points to Line 13 and the red arrows points to Line 4. This means that main called A, but it did **NOT** pass yet the actual parameters 25 and 5. This is supported visually on the right half of the screen a box in bluish gray color labeled as A which means that A is the currently active function. The formal parameters x and y are currently garbage – which means the actual parameters have yet to be passed from main to A.

   So again, please take note Step 4 only does a function call. It will not perform parameter passing yet. The actual parameter passing will be done in the next step – that's the reason why the red arrow is pointing to Line 4.

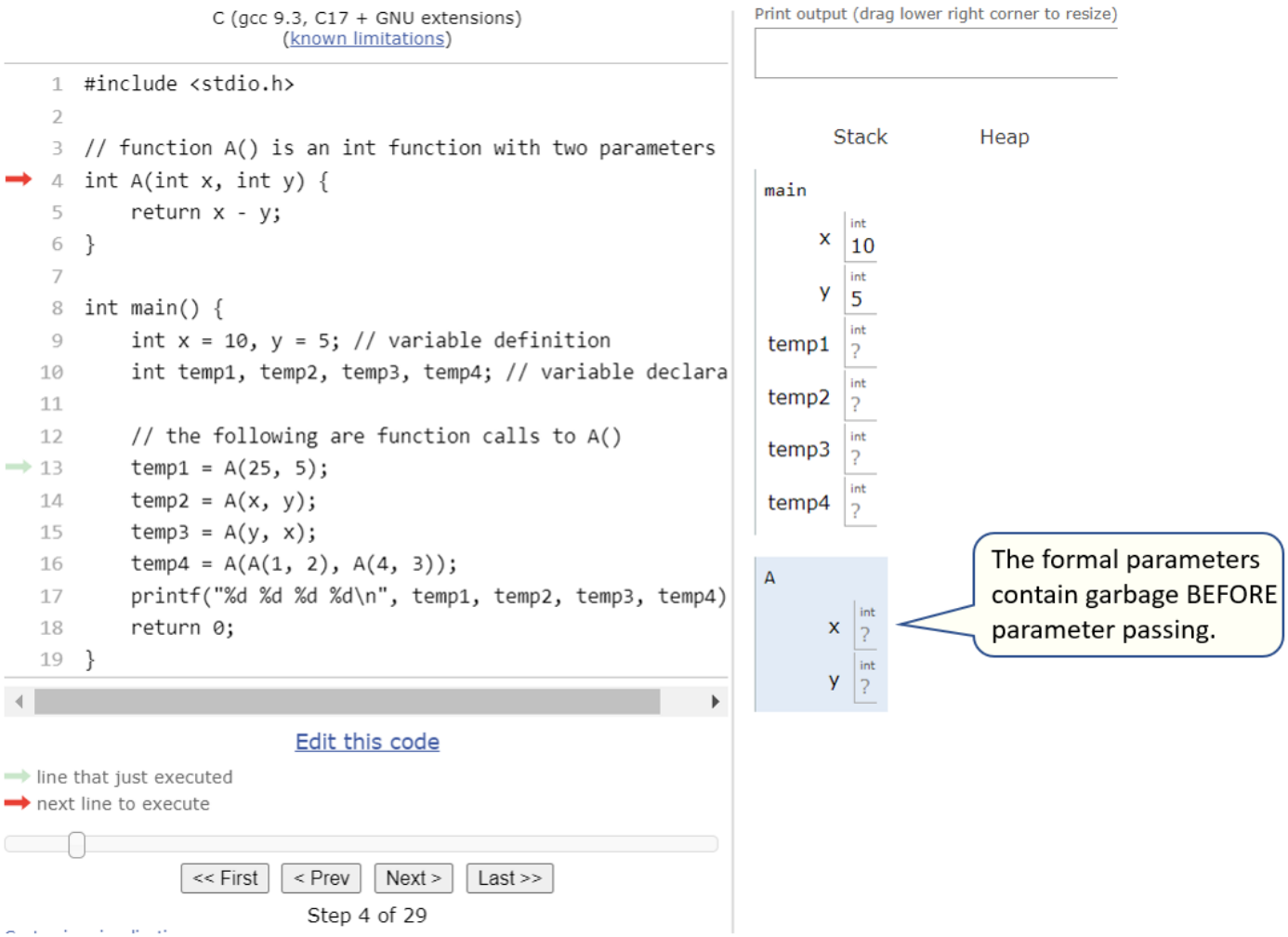

Figure 3.9: main called A; the actual parameters 25 and 5 were NOT yet passed to A's formal parameters x and y respectively.

4. Press the “Next” button again. We’re now in Step 5. Figure 3.10 shows the visualization right after parameter was passing was performed, i.e., the actual parameters 25 and 5 were **copied** as the values of formal parameters x and y. This is visualized on the right half of the screen where the formal parameters now contain 25 and 5 respectively. The next line to execute is Line 5.
5. Press the “Next” button again. We’re now in Step 6. The value to be returned which is 25 – 5 = 20 was computed. Note that the value of 20 is not actually returned now since A is not finished yet. This is shown visually by the red arrow pointing to Line 6.

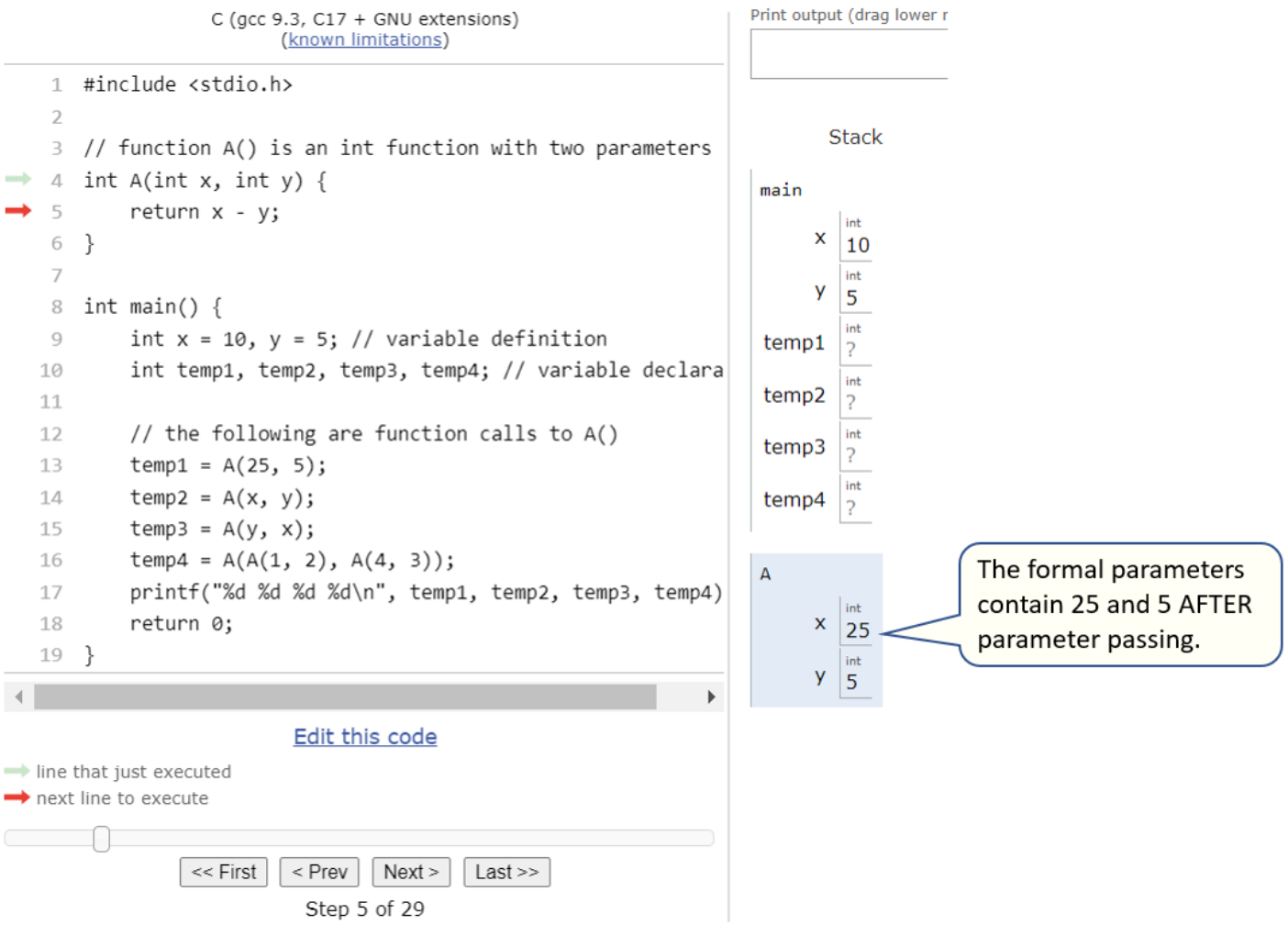

Figure 3.10: Visualization right after passing was performed; formal parameters x and y now contain 25 and 5 respectively.

6. Press the “Next” button again. We’re now in Step 7. Figure 3.11 shows the visualization right after function A finished its task and returned a value of 20 which is assigned as the value of variable temp1. The function main is now active again.

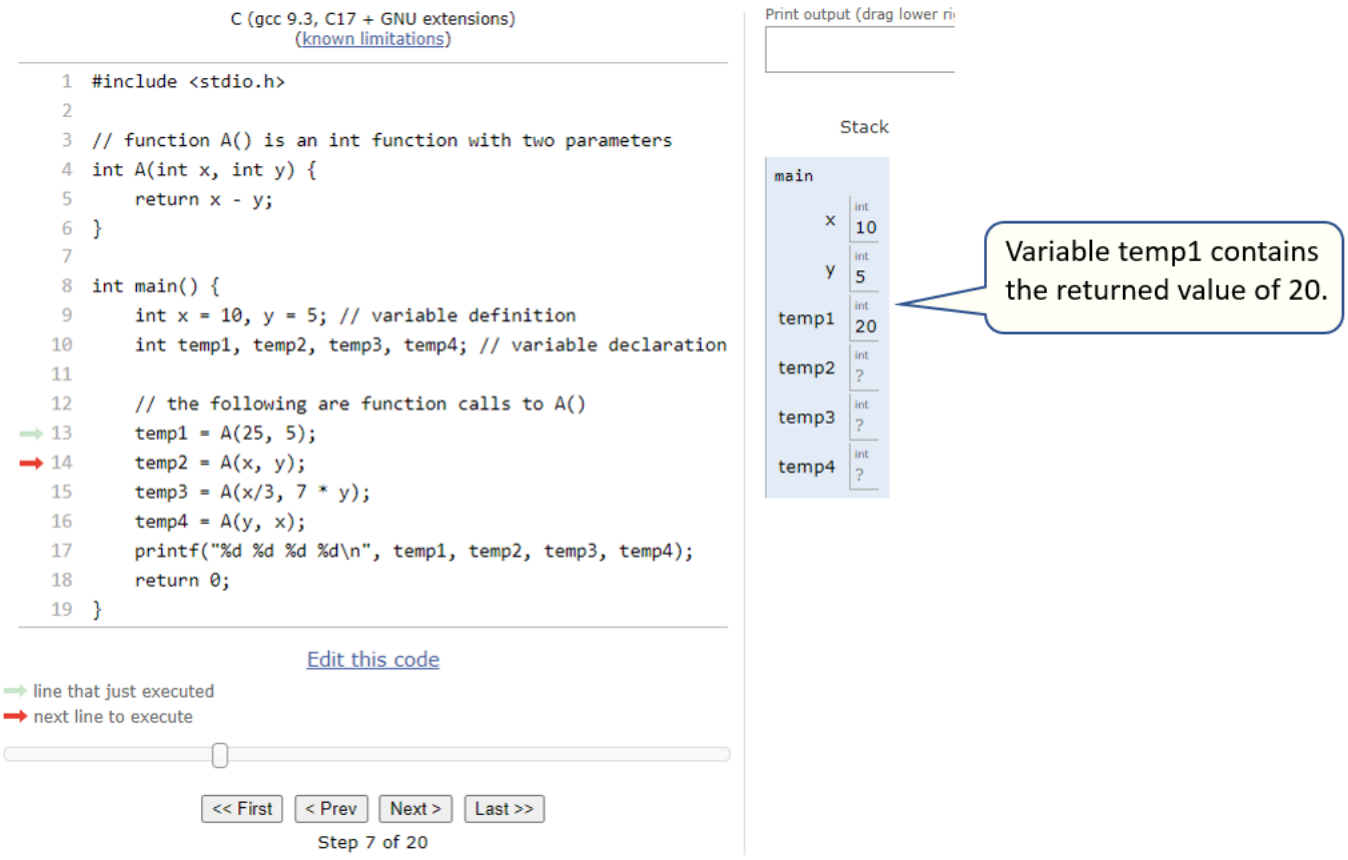

Figure 3.11: A has finished its task and the return value of 20 was assigned (copied) to temp1.

7. Let's move faster a little bit… Press the "Next" button several more times to visualize the effect of the function call and parameter passing in Line 14. Pay special attention to the visualization in Step 10 as shown in Figure 3.12.

   VERY IMPORTANT: There are variables x and y which are local and accessible only inside main. The formal parameters x and y of function A are also local and accessible only inside A. You need remember that although the variable names are the same, please take note that they are different from each other since they are allocated in different memory spaces! This means that whatever you do with the x and y inside A will not affect the values of the x and y inside main. We'll prove this in the next module on visualization of pointers.

   Press "Next" button to step forward and see that the variable temp2 contain a value of 5 which is the value returned by the second function call to A.

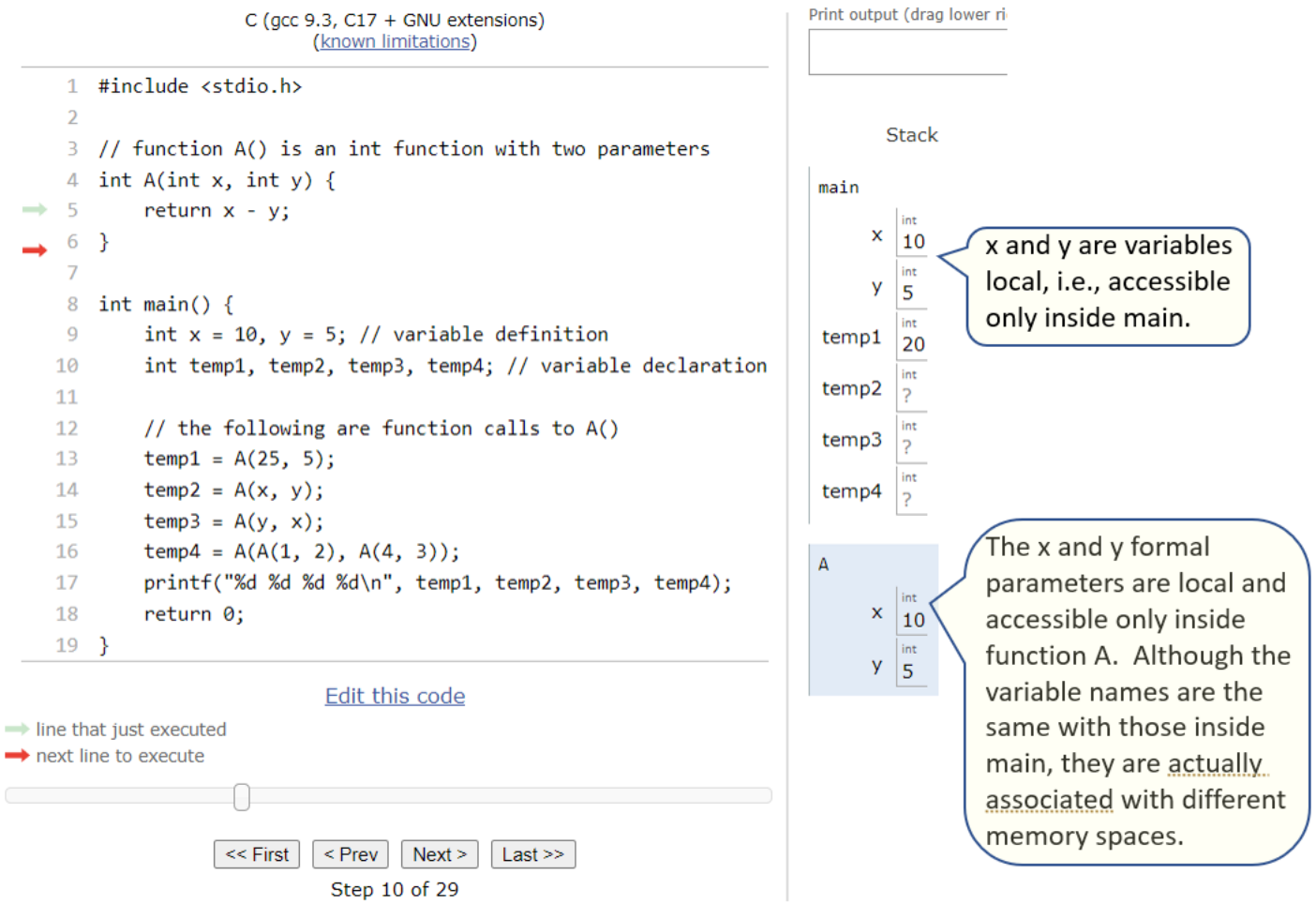

Figure 3.12: Although the variables x and y in main and A have the same name, they are different from each other!

8. Press the “Next” button again until you see Step 12. The green arrow points to Line 15 which means that main called function A. The next step is to pass the actual parameters which are the VALUES of y and x respectively. Take note again of the sequence, the value of y which is 5 will be passed as the first parameter, and then the value of x which is 10 will be passed as the second parameter.

   VERY IMPORTANT: the sequencing of the parameters in the function call MATTERS!

9. Press the “Next” button again. We’re in now Step 13. The visualization should show that formal parameters x and y contain 5 and 10 respectively. Press the “Next” button several more times until you get to Step 15 which shows that temp3 contains the return value of 5 – 10 = -5 as shown in Figure 3.13.

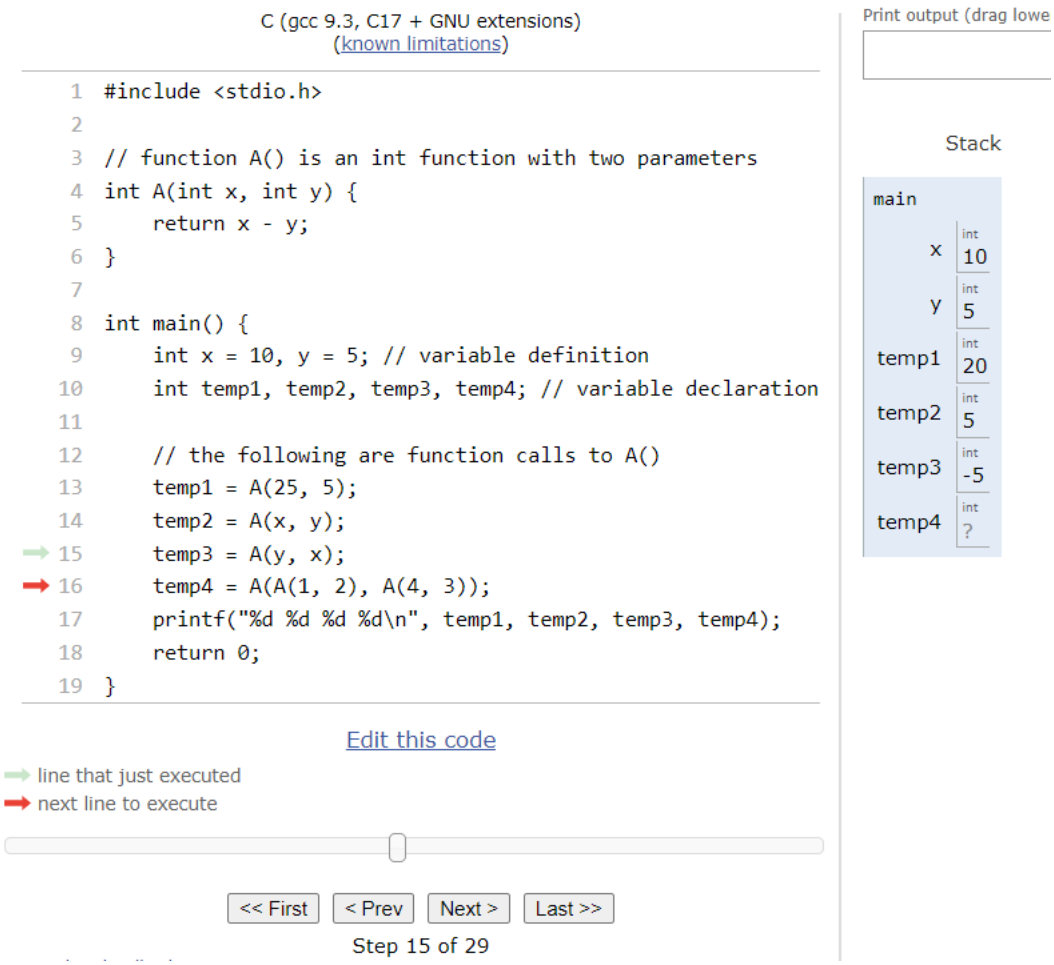

Figure 3.13: Variable temp3 contains -5 after executing the function call in Line 15.

10. Press the "Next" button again. We're now in Step 15 and the next Line to execute is temp4 = A(A(1, 2), A(4, 3)); This is a very "interesting" line of code because it involves three functions calls to A. We know in advance, even before executing the code, that there is an outer function call, and there are two inner function calls A(1, 2) which results to 1 – 2 = -1, and A(4, 3) which results to 4 – 3 = 1. These return values then serve as parameters to the outer function call, i.e., A(-1, 1) which in turn will return -1 – 1 = -2. Question: Which of the two function calls A(1, 2) or A(4, 3) will be executed first? Continue with the visualization to find out.

    Note that this sequencing cannot be seen when you run your program via your DEV-CPP, VSC or XCode IDEs because we really don't have to know and be bothered by the actual details. There is a merit, however, in looking behind the scene and understanding what actually happens during functions calls and parameter passing of "interesting" lines of codes such as temp4 = A(A(1, 2), A(4, 3));.

    Press the "Next" button several times to see how C Tutor executes and visualize Line 16.[2] Please observe carefully what happens in each step. Figure 3.14 shows the visualization after executing the printf statement in Line 17.

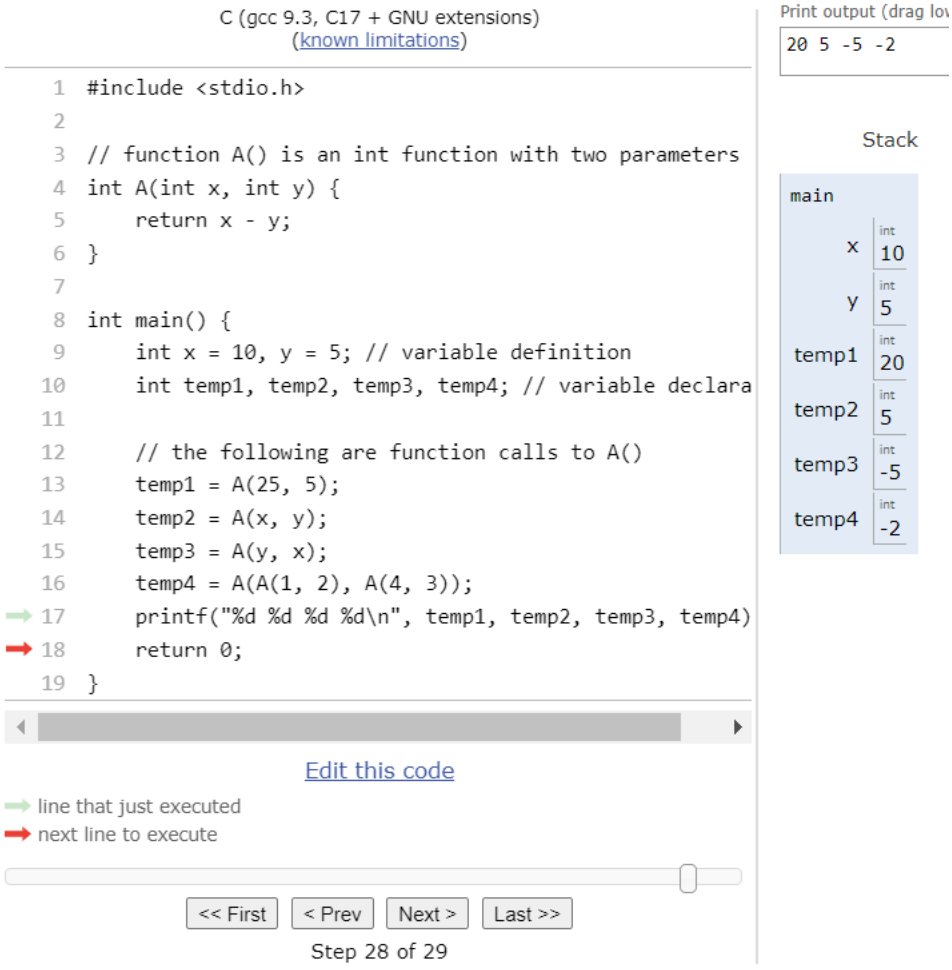

Figure 3.14: Screenshot of the variable values and the printf result

--- End of Module 3. Up Next: Tracing and visualizing pointers. --

[2] Disclaimer: If you did the tracing, you'll find out that C Tutor actually calls and executes A(4, 3) before A(1, 2). Please note that this may not necessarily be the same sequence by which other compilers will sequence the computation steps.