
Tutorial - Debugging and recursion

Some basic coding rules

- ▣ **INDENTATION IS ONE OF THE MOST IMPORTANT PART OF A CODE!!**
- ▣ Have meaningful variable names
- ▣ Break your code into smaller chunks and compile each chunk one by one
- ▣ **Major Aim:** Each chunk should compile and execute successfully.

Errors & Bugs

- **Error** – A mistake in programming is called an error.
 - For example: missing out on completing brackets, typos, missing ';' etc.
- **Bug** – Any unexpected problem which can occur in hardware or a software. Usually occurs as a result of a coding error that could be permanent.
 - For example: A function which calculates the sum which ideally should calculate difference.

Different types of Errors

- ▣ Syntax Errors – indicated by the compiler itself
- ▣ Semantic Errors – incorrect usage of inbuilt statements
- ▣ Logical Errors – Hardest to detect
- ▣ Runtime Errors – program is syntactically correct and gets compiled correctly but its execution leads to abnormal behaviours. (eg: divide by 0 etc)
- ▣ Latent Errors (eg: $(a-b)/(c-d)$, but c becomes equal to d)

Syntax Errors

- Syntax errors occur when the rules of the C language are broken.

Common Causes:

- Missing semicolons
- Incorrect use of brackets or parentheses
- Misspelled keywords
- These errors prevent the code from compiling.

```
#include <stdio.h>
```

```
int main() {
```

```
  Int x = 5 //incorrect identifiers
```

```
  printf("Value of x: %d", x);
```

```
  //Error: Expected ';' before 'printf'
```

```
  printf("Value of x: %d", x;
```

```
  // Error: Expected ')' before ';'
```

```
  return 0;
```

```
}
```

Some
common
syntax
errors

Logical Errors

- Logical errors occur when the program compiles and runs but produces incorrect results.

Common Causes:

- Using incorrect operators
- Wrong calculations
- Incorrect logic in loops or conditions

```
int main() {  
    int a = 5, b = 3;  
    int avg = a + b / 2; // Incorrect order of operations  
    // This is equivalent to a + (b/2)  
    // The correct syntax should be (a+b)/2  
    printf("Average: %d\n", avg);  
  
    // Logical error as the condition always equates to true  
    if(avg = 5){  
        printf("Average is 5\n");  
    }  
    else{  
        printf("Average is not 5\n");  
    }  
  
    return 0;  
}
```

Some
common
logical
errors

Semantic Errors

- Code follows syntax rules but is **incorrect in meaning** due to improper use of language features.

Common causes:

- Using int where float is required
- Implicit type conversion

```
int main() {  
    int num = 10;  
    printf("Value: %f\n", num);  
    // Semantic Error: %f expects a float, but num is an int  
  
    int a = 10, b = 3;  
    float result = a / b;  
    // Semantic Error: Integer division used, discarding decimal part  
    return 0;  
}
```

Some
common
semantic
errors

Runtime Errors

- Runtime errors occur while the program is executing and may cause it to crash.

Common Causes:

- Division by zero
- Accessing out-of-bounds memory
- These errors are detected only when the program is running.

```
int main() {  
    int a = 5, b = 0;  
    int result = a / b;  
    // Runtime error (division by zero)  
    printf("Result: %d\n", result);  
    return 0;  
}
```

Some
common
runtime
errors

Removing Syntax and Semantic Errors

- A syntax error can be easily removed by finding the part of code which is incorrectly written and then correcting the error there.
 - How do we find where exactly is a particular error? – The compiler does that for us.
- For a semantic error, the best way to remove it is to read and understand the code line by line

Removing Logical Errors

- As mentioned, these are the hardest to find.
- Prerequisites before actually starting to write code:
 - Read the problem statement slowly at least twice.
 - Take a pen and paper and list down the requirements of the problem statement
 - Devise a rough algorithm for the problem given.
 - Build logic for different subparts of the problems differently

THE BIGGEST CHALLENGE

- The program works, but the output is not as desired.
- This simply means that the logic devised for the problem is either completely incorrect or partially incorrect
- Clue to find what is wrong: check the flow of control and values of variables in the code
- How to fix this:
 - Try to change the logic (if feasible)
 - If you feel the logic is correct, give a Dry Run to your code
 - Try to find the minimum snippet of the code that results in wrong output

Debugging and its Necessity

- Basic definition: finding out what is not working and fixing it
- Systematic process of identifying and removing existing errors in a particular piece of code
- Usually is more challenging than writing the code itself.
- **A PART OF WRITING CODE!!**
- Testers just tell you that your code fails, not why it fails.

Dry Run of a code

- One of the best ways of getting the “result” of a code, without execution!
- During dry run, the programmer acts like a compiler and executes the code line by line by taking a random sample input to the problem
- In the simplest terms, dry run is the execution of your code on pen and paper
- USEFUL IN ELIMINATING ALMOST EVERY TYPE OF ERROR

Steps to Debug your Code

STEP 0: Don't Panic on seeing an error!

- Bugs are not always complex. Fixing a simple typo can also change the output completely
- Changing the code might work in some cases, but mostly it doesn't
- Don't ever make changes without a logic or plan

STEP 1: Identify the specific issue you have

- For example – identify what type of error is it
- According to the type of error, try to find a solution to fix it.

STEP 2: Determine the root cause of the error

- Localise the problem into different parts
- For example – If a program has different functions, which function exactly gives the error

Some General Tips

- ❑ Always read and try to understand the error carefully.(You cannot be a good programmer unless you have the courage to **generate and fix** errors)
- ❑ In case of syntax errors, wherever you get an error, always check the line written before and the line written after the particular line where you get an error.
- ❑ Run the code multiple times (even if you just add a print statement)
- ❑ Use comments as much as possible
- ❑ Don't just build a code based on sample test cases. Try to build a generalised version.
- ❑ If your code works for sample cases, always create a border test case and dry run your code on that case.

Some General Tips (contd.)

Although a systematic approach should be followed, sometimes, intuitions and beliefs work great

Never be afraid to use variables.

Always be careful while fixing a bug, as the following always happens:



- **BE CRITICAL OF YOUR CODE**
- **ALWAYS INDENT YOUR CODE PROPERLY**
- **BE PATIENT!**

Using print statements to debug

Using VSC Debugger

Click on those lines of code where you would like to monitor the values of the variables and add breakpoints.

Use CTRL+SHIFT+D to open the debugging window.

Then use F5 to start debugging.

3. Click on Run and then "Start Debugging" from the drop down menu.

1. Add breakpoints in the code

2. Click on the Run and Debug option in the command palette (or press CTRL+SHIFT+D)

```
DebuggingLab.c
1  #include<stdio.h>
2
3  int main(){
4      printf("Hello World!\n");
5      int a = 0;
6      if (a == 1){
7          a += 3;
8      }
9      if (a || !a){
10         a += 2;
11     }
12     else a = a + 2;
13
14     return 0;
15 }
```

```
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL
Hello World!
[1] + Done
"/usr/bin/gdb" --interpreter=mi --tty=${DbgTerm} 0<"/tmp/Microsoft-MIEngine-In-jte50ip5.twt" 1>"/tmp/
Microsoft-MIEngine-Out-i0t3Uwcj.w5s"
dhruthi30@LAPTOP-7KN15LKH:~/programs$
```

RUN AND DEBUG

Debug Program

VARIABLES

WATCH

CALL STACK

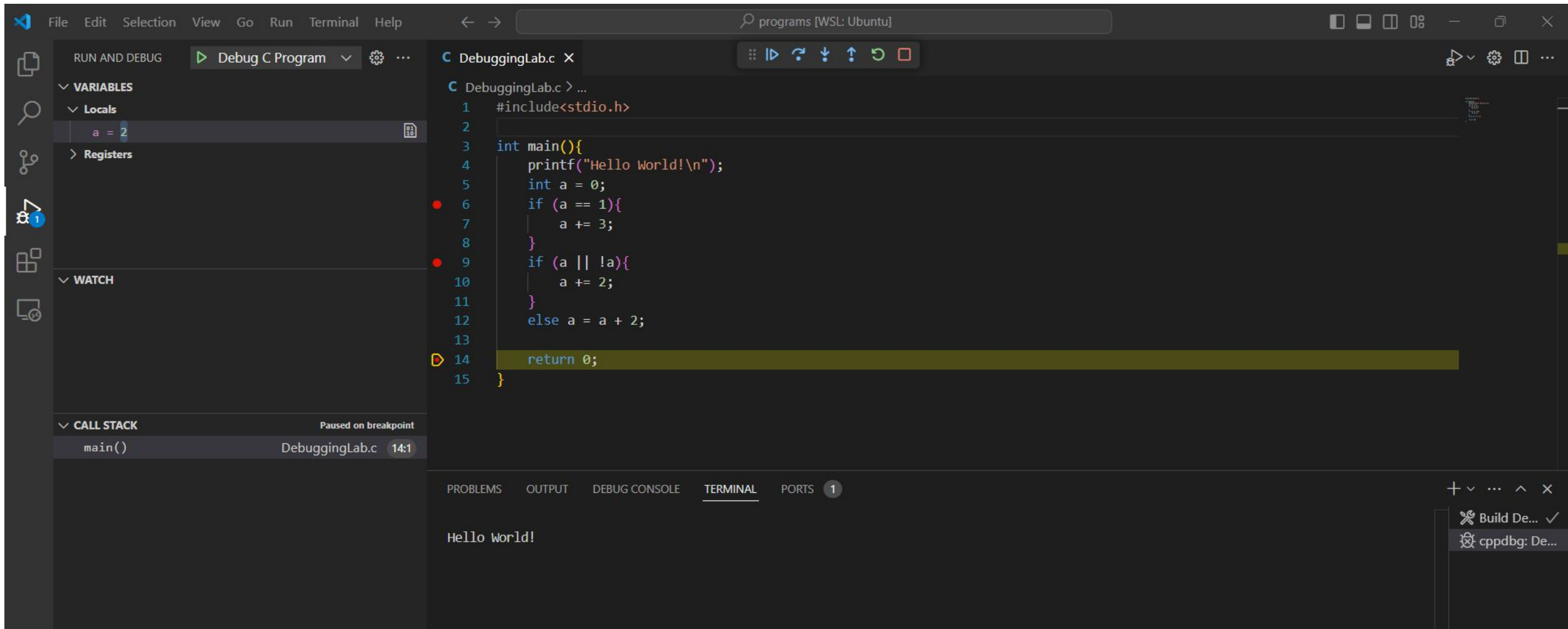
BREAKPOINTS

All C++ Exceptions

DebuggingLab.c

DebuggingLab.c

DebuggingLab.c










Your screen should look something like this.

Using the VSCode Debugger

Terminology

There are some terms you'll need to be familiar with before jumping-in. You'll see practical demonstrations soon.

-  **Breakpoint** - A marker set by the programmer at a particular line of code that indicates where program control should halt until the next command dictated by the programmer.
- **Step/Stepping** - Executing the program line-by-line.
-  **Continue** - A command that runs program execution until the next breakpoint, or until the program ends.
-  **Step Over** - A command that executes one line of code without going into any function that line might be invoking.
-  **Step Into** - A command that executes one line of code and goes into any function that line might be invoking.
-  **Step Out** - A command that runs the function when control has stepped into one, bringing control back to the line that called the function via its return.
-  **Restart** - A command that re-executes the program from the beginning with the debugger active.
-  **Stop** - A command that exits the debugger and halts program execution entirely.

Recursion : Abstraction of a problem

Definition:

Recursion is a technique where a function calls itself to solve a problem.
Each recursive call breaks the problem down into smaller instances.
A recursive function must have a base case to stop infinite recursion.

Consider the following problem:

You are standing in a queue in a straight line. You cannot see anyone other than the person directly in front of you (Assume this is true for everyone, i.e every person can only see the person in front of them.) You know you are the last person in the line. How do you figure out how long you'd have to wait?

When do we use recursion?

- Suppose there is a problem which you are not sure how to solve, but you can break it down to an identical problem of smaller size.
- Write code under the assumption that you have the solution to the smaller sized problem.
- Take care of the base case, i.e when the problem can no longer be broken down.

You ask the person in front of you how many people there are in front of him. He asks the same question to the person in front of him. Recursive call

This goes on until you reach the front of the queue. The person at the front answers, "Zero". Base case

Now this answer propagates backward, and each person adds one to account for the person standing in front of them. Return statements.

Now you know the number of people in line!

Recursion as function calls

- Each function call is pushed onto the call stack.
- The function executes until the base case is reached.
- After reaching the base case, the stack unwinds (returns values step by step).

Factorial of a number using recursion

```
factorial(5)
```

```
├─ 5 * factorial(4)
│   ├─ 4 * factorial(3)
│       ├─ 3 * factorial(2)
│           ├─ 2 * factorial(1)
│               ├─ 1 * factorial(0)
│                   └─ Return 1 (Base Case)
│               └─ Return 1 * 1 = 1
│           └─ Return 2 * 1 = 2
│       └─ Return 3 * 2 = 6
│   └─ Return 4 * 6 = 24
└─ Return 5 * 24 = 120
```

```
#include <stdio.h>
```

```
int factorial(int n) {
    if (n == 0) return 1; // Base case
    return n * factorial(n - 1); // Recursive call
}

int main() {
    int num = 5;
    printf("Factorial of %d is %d\n", num, factorial(num));
    return 0;
}
```

Lastly, A general coding tip

**ALWAYS CODE
AS IF THE GUY
WHO ENDS UP
MAINTAINING,
OR TESTING
YOUR CODE
WILL BE A
VIOLENT
PSYCHOPATH
WHO KNOWS
WHERE YOU
LIVE.**

~dave carhart

Thanks

