SE Assignment 3

Name: Himani Verma

Admission NO: U19CS075

1. Implement the following problematic control structures in C and compare the outputs of standard C compiler and the Splint tool.

* Likely infinite loops

Source Code:

#include <stdio.h>

void infy(){

    int i = 1;

    while(i<10)

    {

        printf("%d\n", i);

    }

}

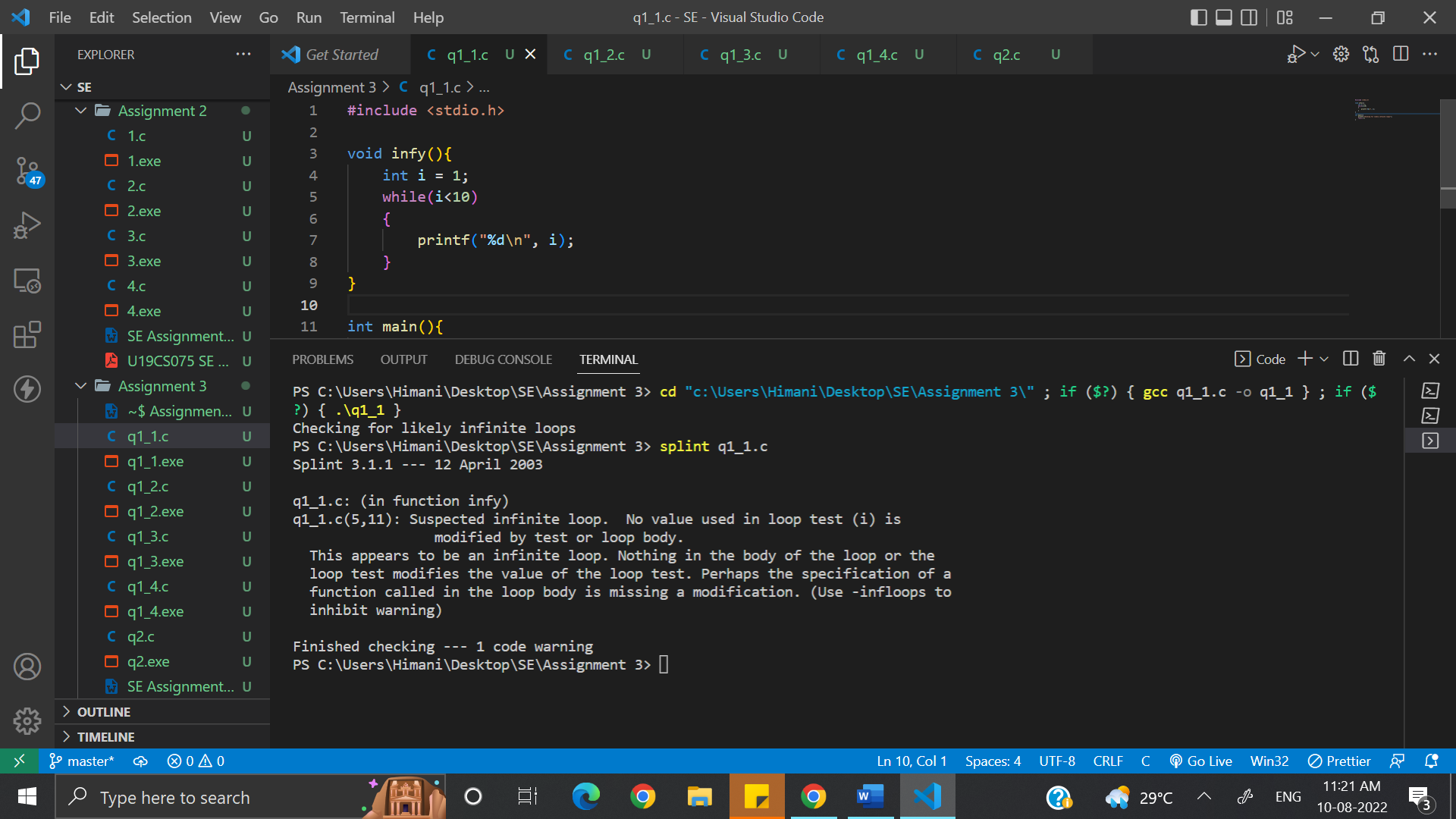
int main(){

    printf("Checking for likely infinite loops");

    return 0;

}

Output:



* Fall through switch cases

Source Code:

#include <stdio.h>

typedef enum{

    YES,

    NO,

    DEFINITELY,

    PROBABLY,

    MAYBE

} ynm;

void decide(ynm y){

    switch (y){

    case PROBABLY:

    case NO:

        printf("No!");

    case MAYBE:

        printf("Maybe");

    case YES:

        printf("Yes!");

    case DEFINITELY:

        printf("Definitely!");

    }

}

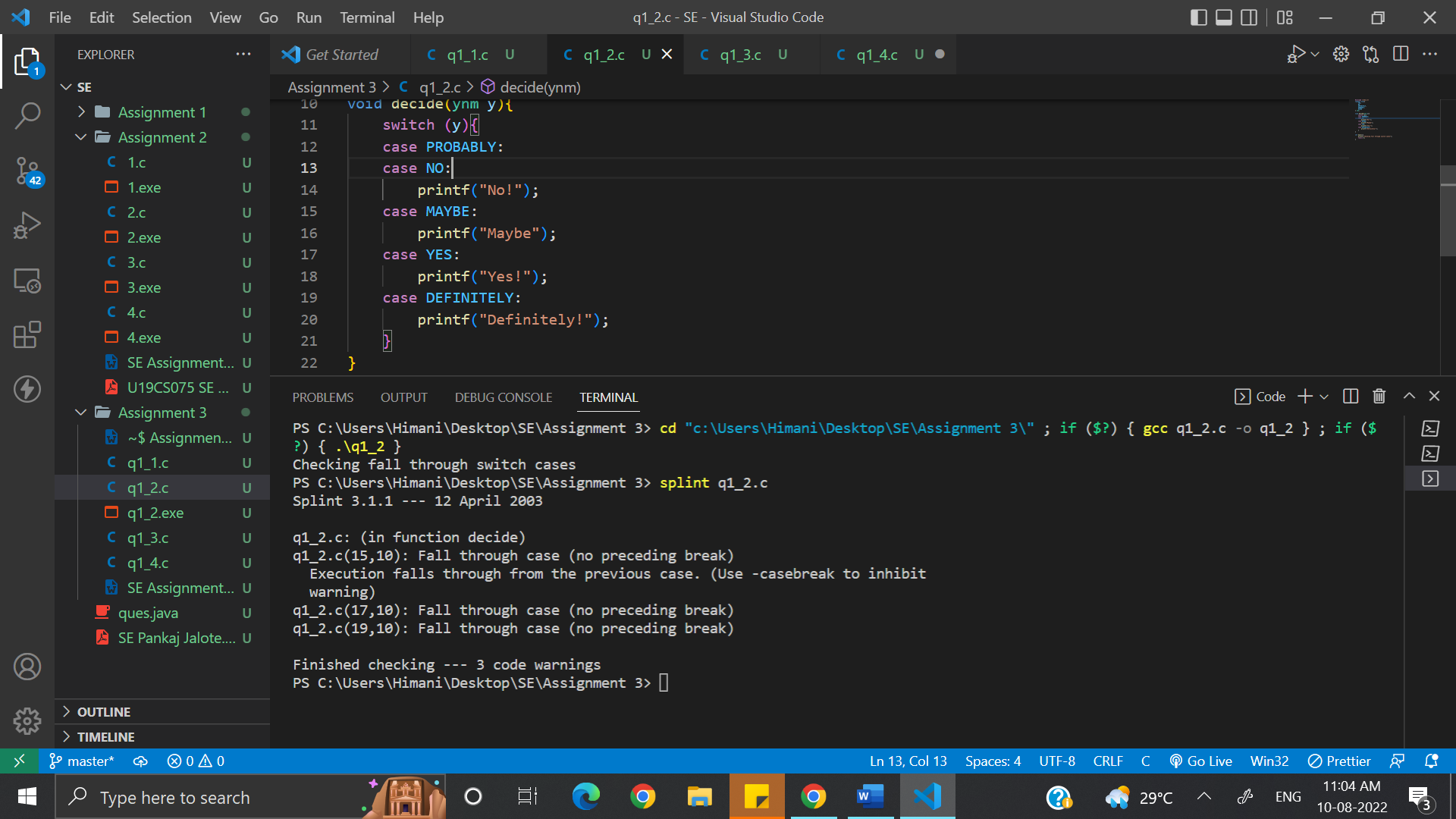
int main(){

    printf("Checking fall through switch cases");

    return 0;

}

Output:



* Missing switch cases

Source Code:

#include <stdio.h>

typedef enum { YES, NO, DEFINITELY, PROBABLY, MAYBE } ynm;

void decide(ynm y){

    switch (y){

    case PROBABLY:

        break;

    case NO:

        printf("No!");

        break;

    case MAYBE:

        printf("Maybe");

        break;

    case YES:

        printf("Yes!");

        break;

    }

}

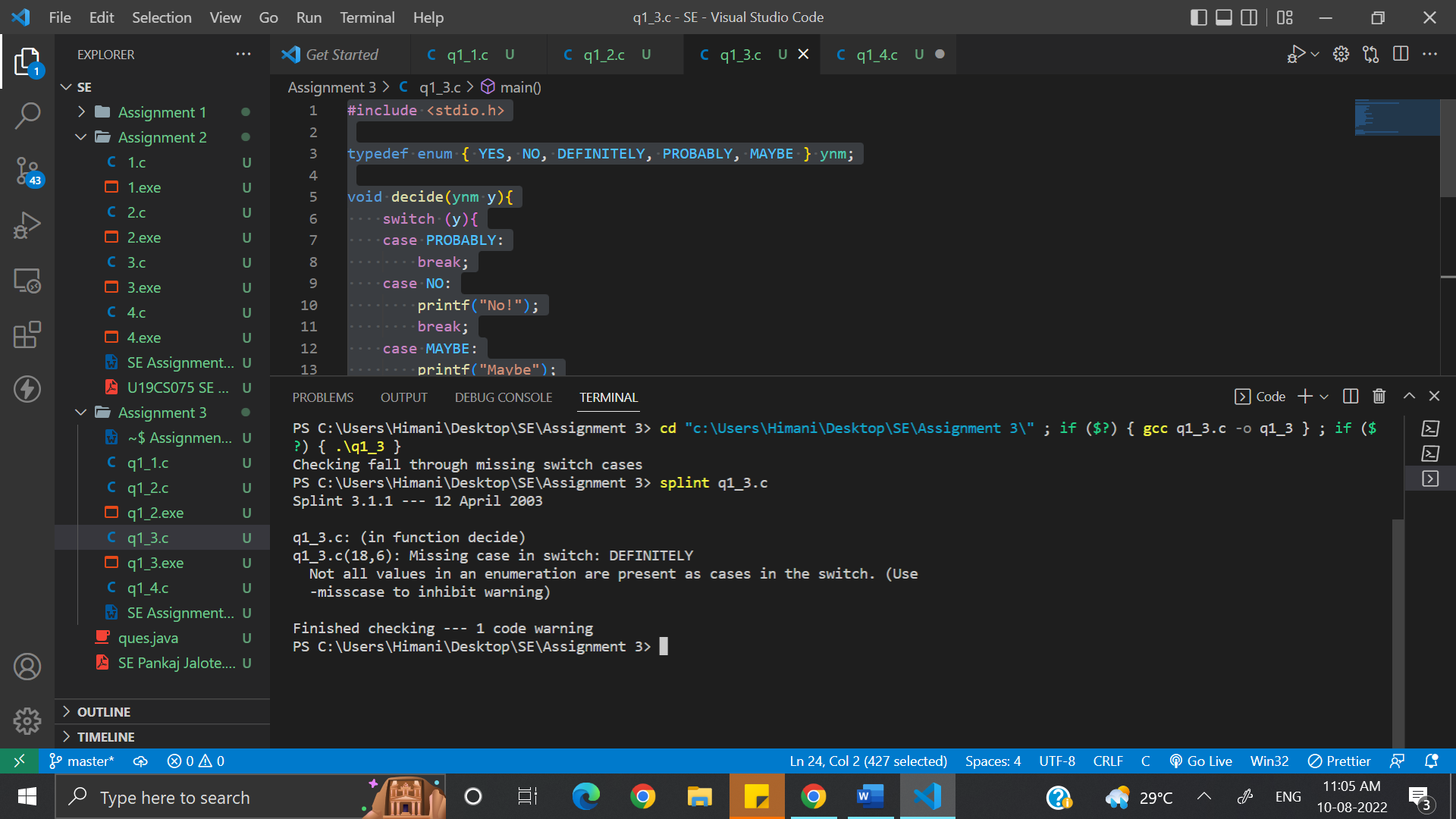
int main(){

    printf("Checking fall through missing switch cases");

    return 0;

}

Output:



* Empty statement after an if, while or for

Source Code:

#include <stdio.h>

int main(){

    int x = 1;

    if (x > 3)

        ;

    if (x > 3)

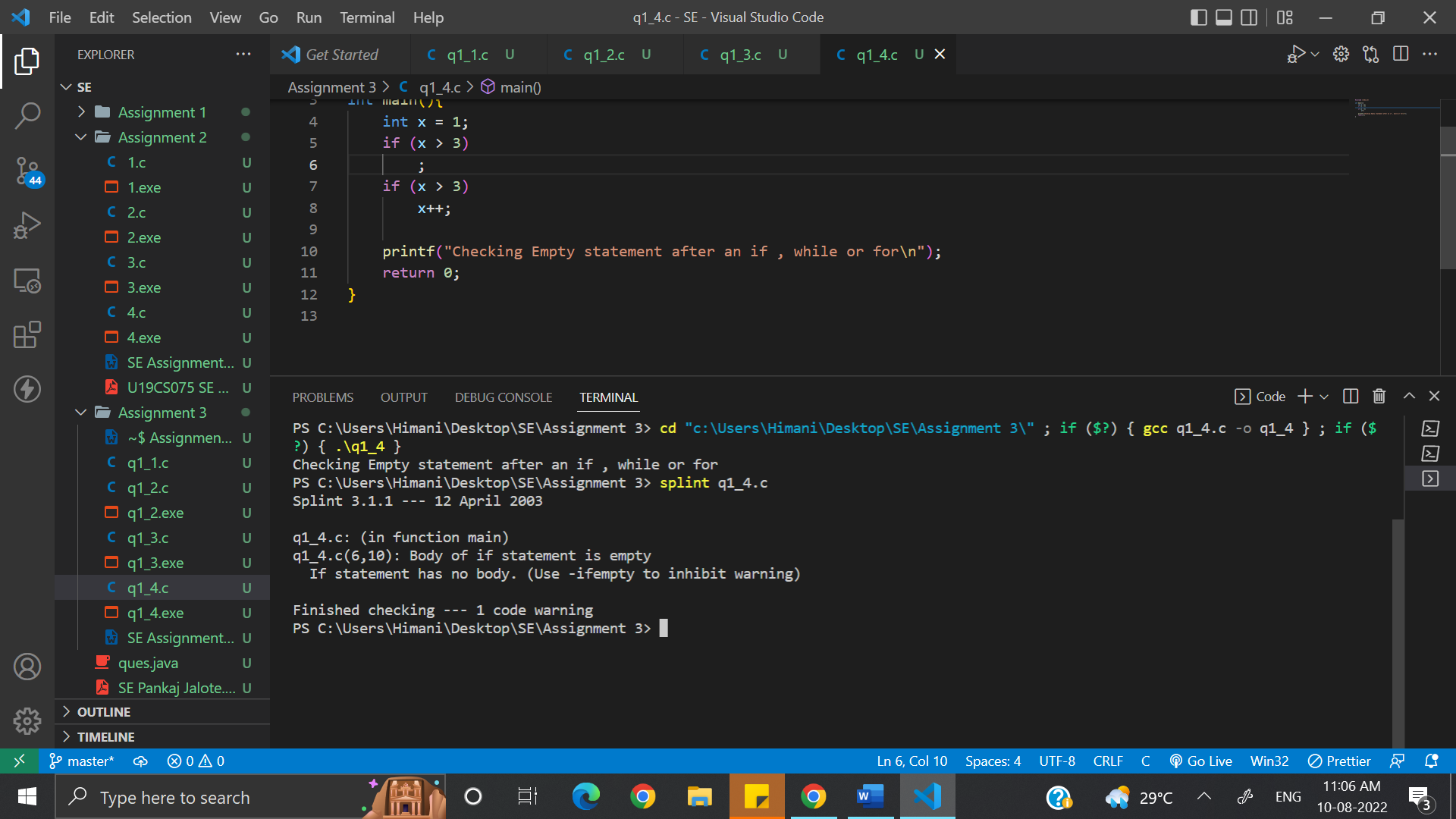
        x++;

    printf("Checking Empty statement after an if , while or for\n");

    return 0;

}

Output:



2. What is buffer overflow? How it can be exploited? Write a C program to illustrate a buffer overflow attack?

Buffer overflow is a software coding error or vulnerability that can be exploited by hackers to gain unauthorized access to corporate systems. It is one of the best-known software security vulnerabilities yet remains fairly common. This is partly because buffer overflows can occur in various ways and the techniques used to prevent them are often error-prone.

The buffer overflow exploit techniques a hacker uses depends on the architecture and operating system being used by their target. However, the extra data they issue to a program will likely contain malicious code that enables the attacker to trigger additional actions and send new instructions to the application.

Source Code:

#include <stdio.h>

#include <string.h>

#include<stdlib.h>

void BufferO() {

    char \*ptr  = (char\*) malloc(10);

    ptr[10] = 'c';

}

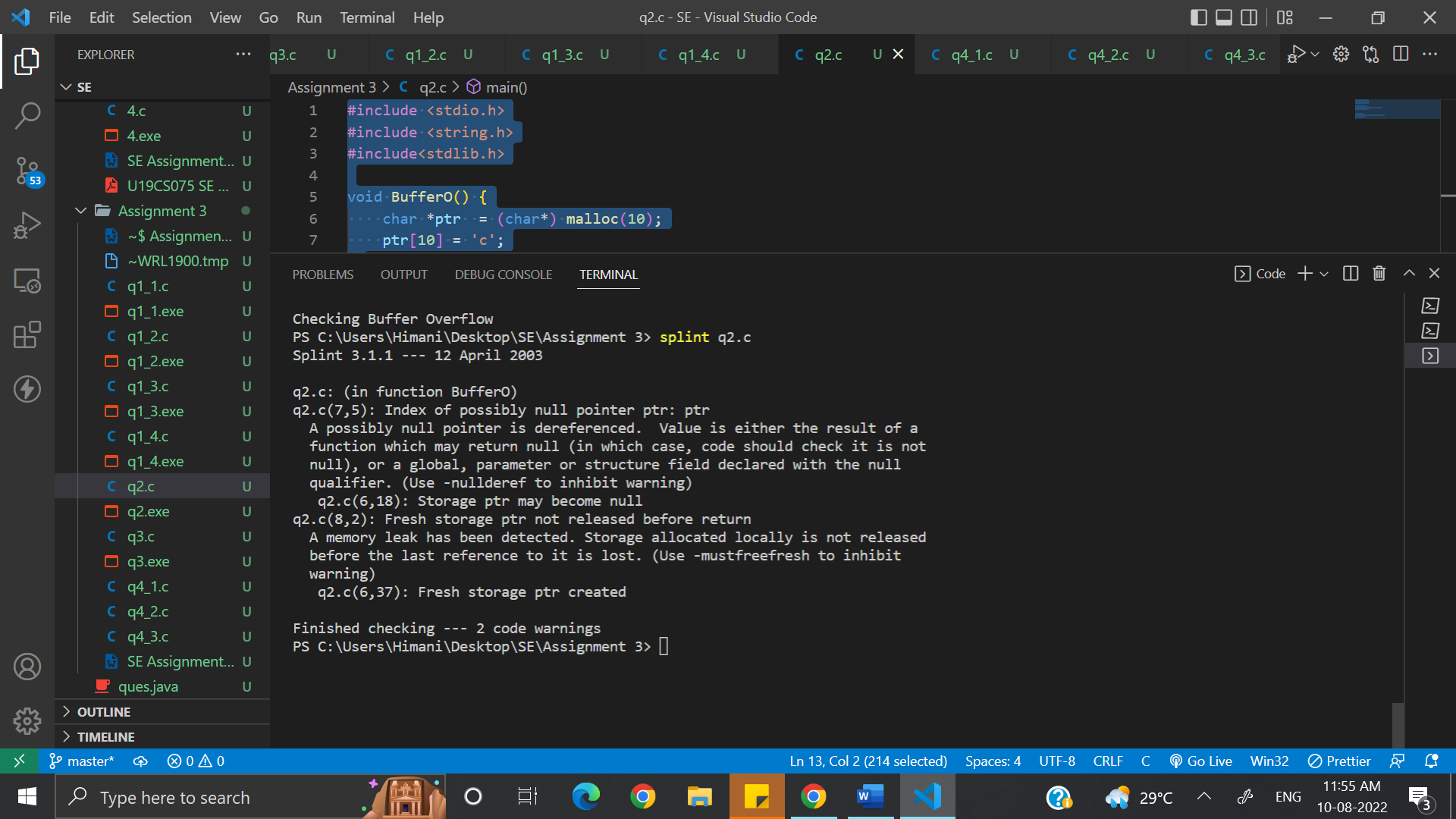
int main(){

    printf("Checking Buffer Overflow");

    return 0;

}

Output:



3. Macro implementations or invocations can be dangerous. Justify this statement by giving an example in C language.

Splint eliminates most of the potential problems by detecting macros with dangerous implementations and dangerous macro invocations. Whether or not a macro definition is checked or expanded normally depends on flag settings and control comments. Stylized macros can also be used to define control structures for iterating through many values.

Source Code:

#include <stdio.h>

extern int square(int x);

#define square(x) ((x) \* (x))

extern int sumsquares(int x, int y);

#define sumsquares(x, y) (square(x) + square(y))

int main(){

    int i = 1;

    i = square(i++);

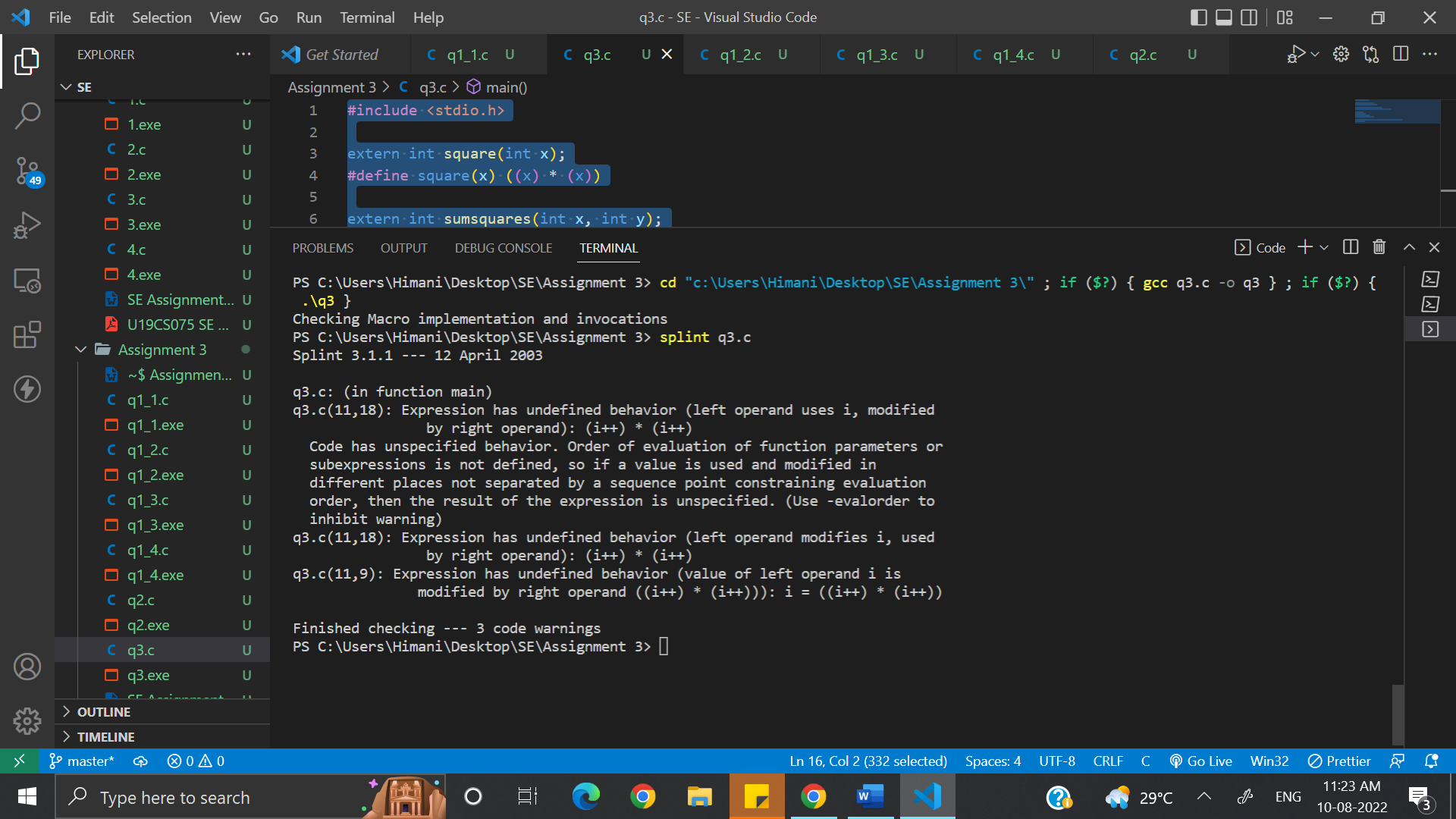
    i = sumsquares(i, i);

    printf("Checking Macro implementation and invocations\n");

    return 0;

}

Output:



4. What do you mean by interface faults. Write a set of C programs to implement interface faults and perform their detection using Splint tool. Check whether they are detected by the standard C compiler or not.

Functions communicate with their calling environment through an interface. The caller communicates the values of actual parameters and global variables to the function, and the function communicates to the caller through the return value, global variables and storage reachable from the actual parameters. By keeping interfaces narrow (restricting the amount of information visible across a function interface), we can understand and implement functions independently.

Declaration:

Source Code:

//Declaration

extern void setx(int \*x, int \*y)/\*@modifies \*y@\*/;

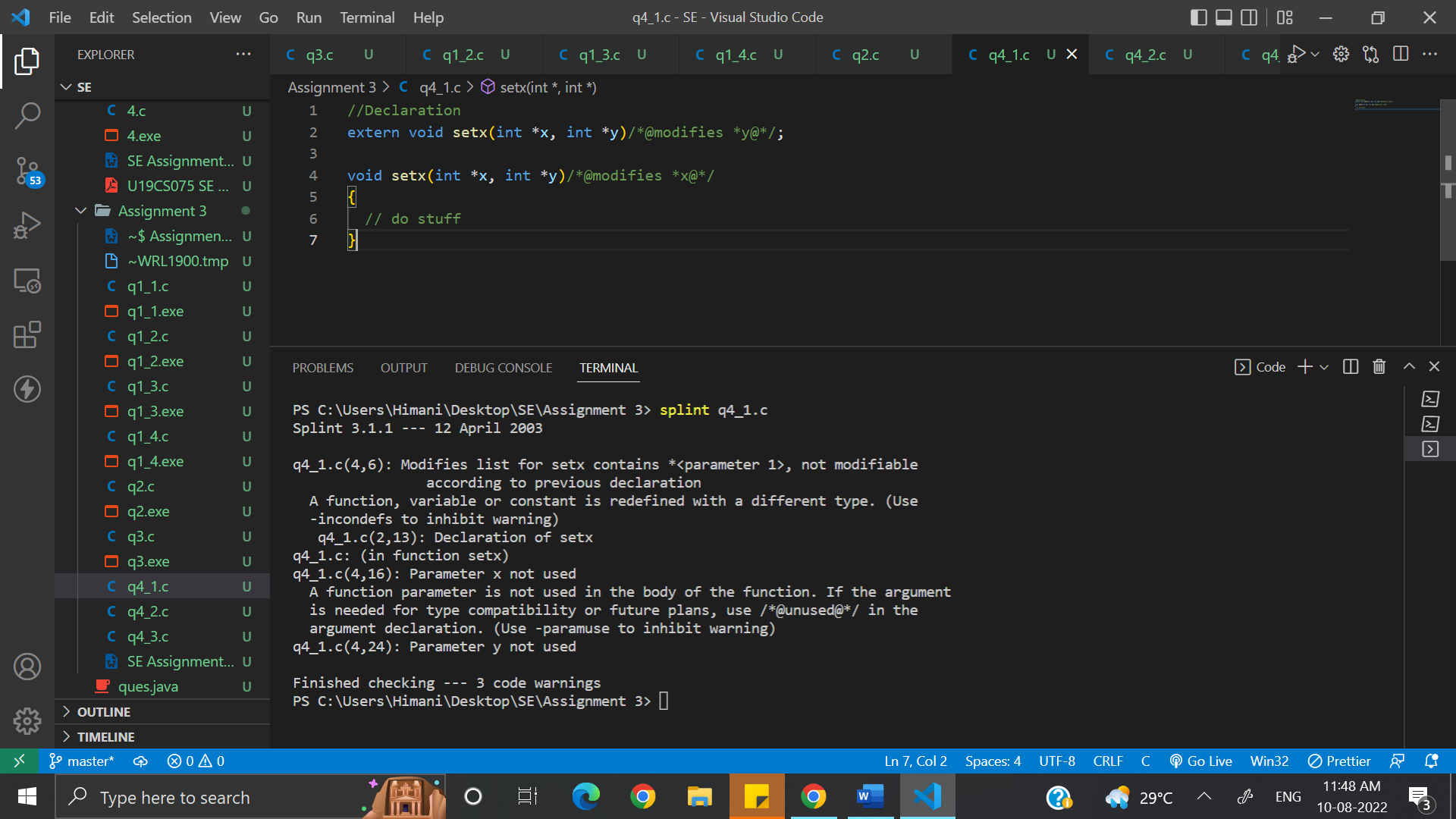
void setx(int \*x, int \*y)/\*@modifies \*x@\*/

{

  // do stuff

}

Output:



Global:

Source Code:

//Global

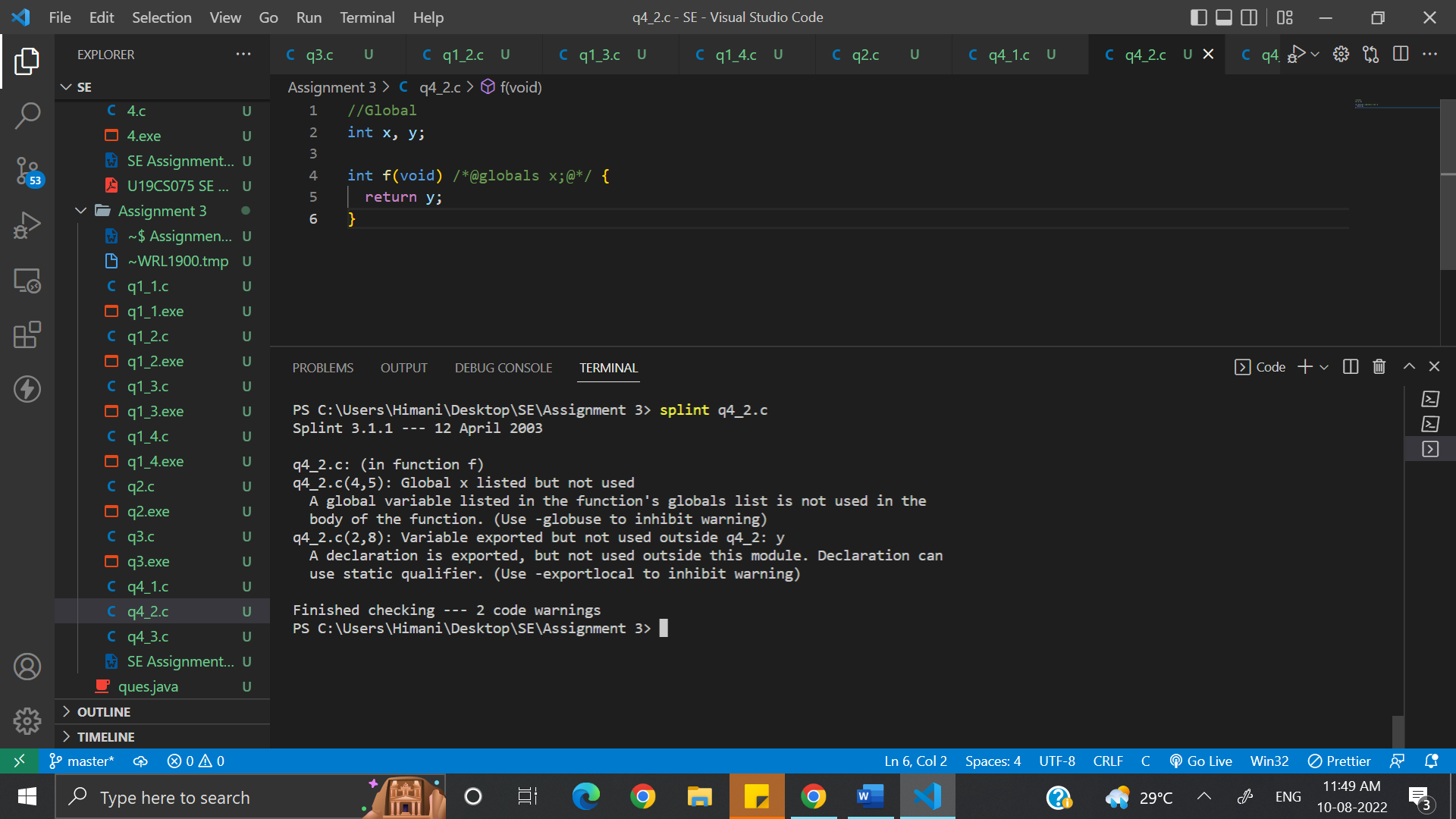
int x, y;

int f(void) /\*@globals x;@\*/ {

  return y;

}

Output:



Modification:

Source Code:

//Modification

void setx(int \*x, int \*y)

/\*@modifies \*x@\*/

{

  \*y = \*x;

}

void sety(int \*x, int \*y)

/\*@modifies \*y@\*/

{

  setx(y, x);

}

Output:

