SE Assignment 3

Name: Himani Verma

Admission NO: U19CS075

---

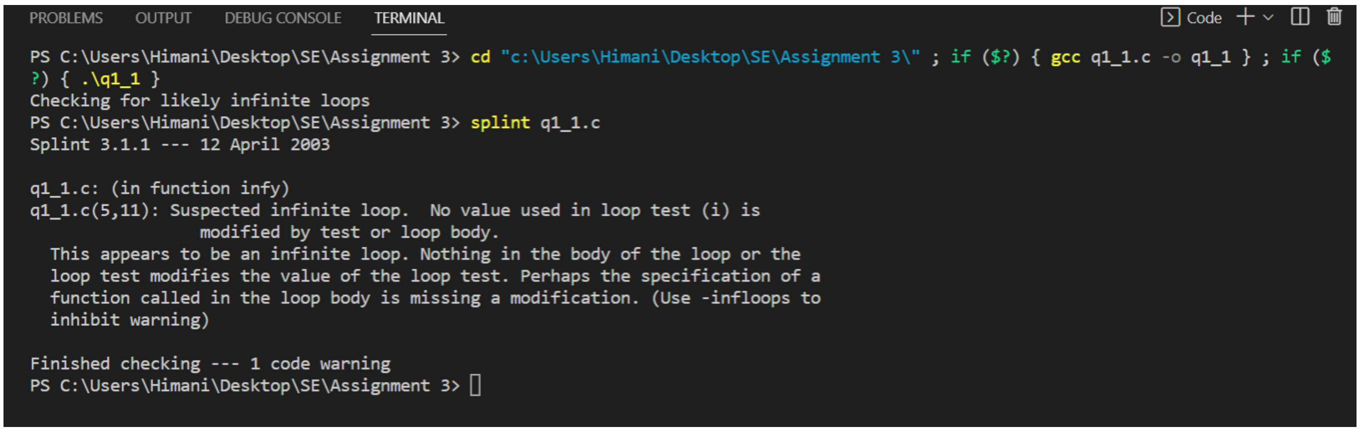
1. Implement the following problematic control structures in C and compare the outputs of standard C compiler and the Splint tool.

- **Likely infinite loops**

Source Code:

```c
#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:



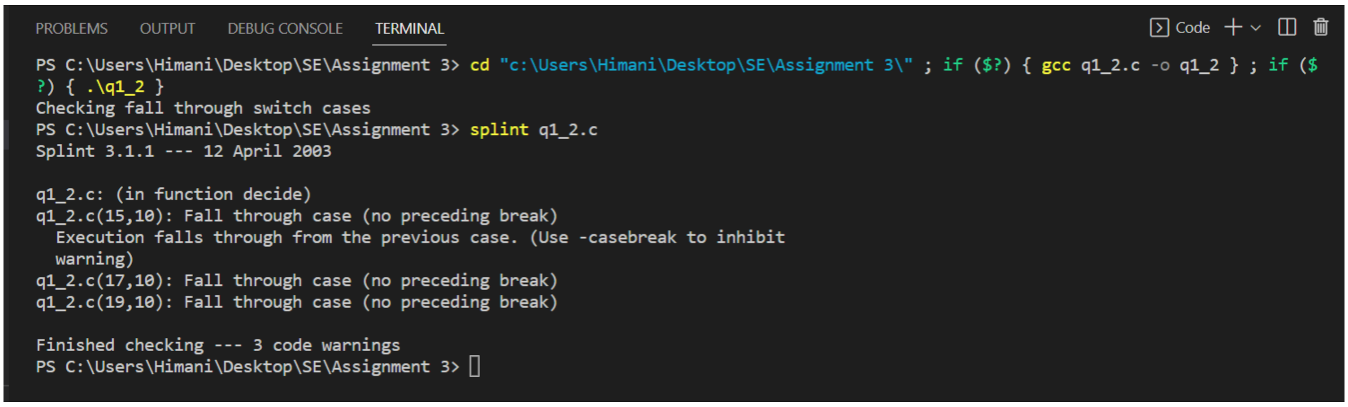
```
PS C:\Users\Himani\Desktop\SE\Assignment 3> cd "c:\Users\Himani\Desktop\SE\Assignment 3\" ; if ($?) { gcc q1_1.c -o q1_1 } ; if ($
?) { .\q1_1 }
Checking for likely infinite loops
PS C:\Users\Himani\Desktop\SE\Assignment 3> splint q1_1.c
Splint 3.1.1 --- 12 April 2003

q1_1.c: (in function infy)
q1_1.c(5,11): Suspected infinite loop.  No value used in loop test (i) is
                 modified by test or loop body.
  This appears to be an infinite loop. Nothing in the body of the loop or the
  loop test modifies the value of the loop test. Perhaps the specification of a
  function called in the loop body is missing a modification. (Use -infloops to
  inhibit warning)

Finished checking --- 1 code warning
PS C:\Users\Himani\Desktop\SE\Assignment 3>
```

- **Fall through switch cases**

Source Code:

```c
#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:



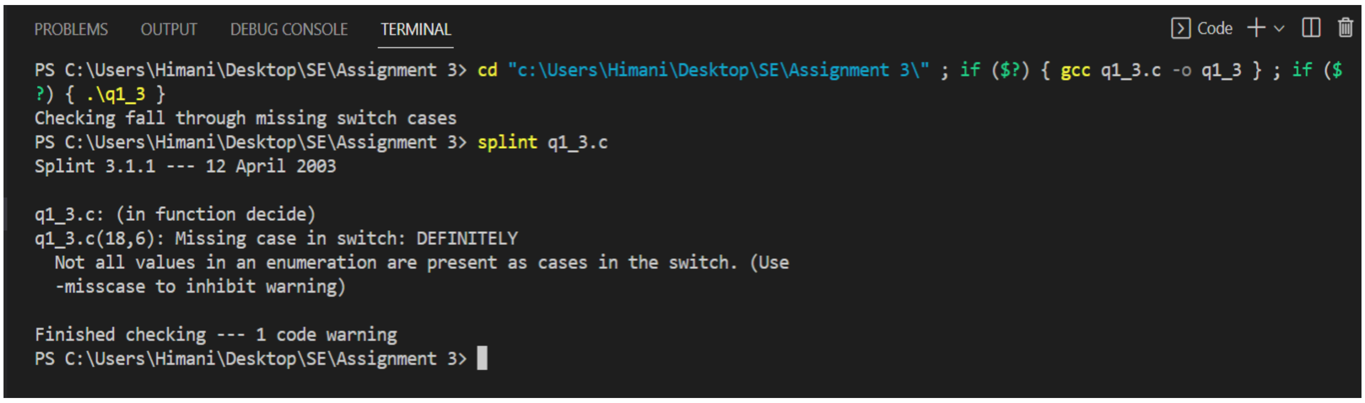
```
PS C:\Users\Himani\Desktop\SE\Assignment 3> cd "c:\Users\Himani\Desktop\SE\Assignment 3\" ; if ($?) { gcc q1_2.c -o q1_2 } ; if ($
?) { .\q1_2 }
Checking fall through switch cases
PS C:\Users\Himani\Desktop\SE\Assignment 3> splint q1_2.c
Splint 3.1.1 --- 12 April 2003

q1_2.c: (in function decide)
q1_2.c(15,10): Fall through case (no preceding break)
  Execution falls through from the previous case. (Use -casebreak to inhibit
  warning)
q1_2.c(17,10): Fall through case (no preceding break)
q1_2.c(19,10): Fall through case (no preceding break)

Finished checking --- 3 code warnings
PS C:\Users\Himani\Desktop\SE\Assignment 3> 
```

- **Missing switch cases**

Source Code:

```c
#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:



PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

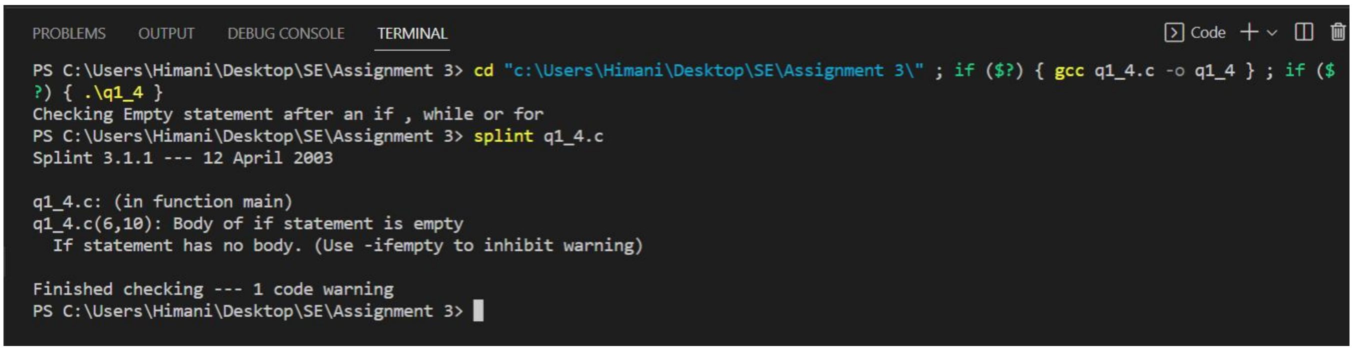
```
PS C:\Users\Himani\Desktop\SE\Assignment 3> cd "c:\Users\Himani\Desktop\SE\Assignment 3\" ; if ($?) { gcc q1_3.c -o q1_3 } ; if ($
?) { .\q1_3 }
Checking fall through missing switch cases
PS C:\Users\Himani\Desktop\SE\Assignment 3> splint q1_3.c
Splint 3.1.1 --- 12 April 2003

q1_3.c: (in function decide)
q1_3.c(18,6): Missing case in switch: DEFINITELY
  Not all values in an enumeration are present as cases in the switch. (Use
  -misscase to inhibit warning)

Finished checking --- 1 code warning
PS C:\Users\Himani\Desktop\SE\Assignment 3>
```

- **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:



```
PS C:\Users\Himani\Desktop\SE\Assignment 3> cd "c:\Users\Himani\Desktop\SE\Assignment 3\" ; if ($?) { gcc q1_4.c -o q1_4 } ; if ($
?) { .\q1_4 }
Checking Empty statement after an if , while or for
PS C:\Users\Himani\Desktop\SE\Assignment 3> splint q1_4.c
Splint 3.1.1 --- 12 April 2003

q1_4.c: (in function main)
q1_4.c(6,10): Body of if statement is empty
  If statement has no body. (Use -ifempty to inhibit warning)

Finished checking --- 1 code warning
PS C:\Users\Himani\Desktop\SE\Assignment 3>
```

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

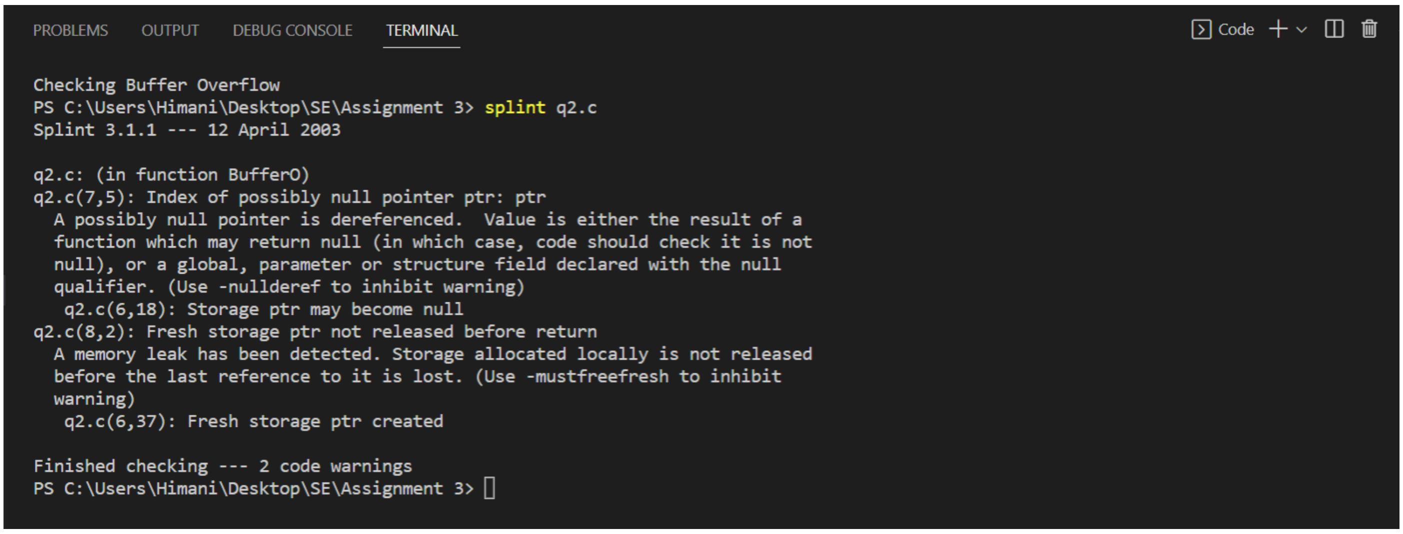
Buffer overflow is a software coding error or vulnerability that can be exploited by hackers to gain unauthorized access to corporate systems. It is one of the best-known software security vulnerabilities yet remains fairly common. This is partly because buffer overflows can occur in various ways and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

```c
#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:



```
Checking Buffer Overflow
PS C:\Users\Himani\Desktop\SE\Assignment 3> splint q2.c
Splint 3.1.1 --- 12 April 2003

q2.c: (in function Buffer0)
q2.c(7,5): Index of possibly null pointer ptr: ptr
  A possibly null pointer is dereferenced.  Value is either the result of a
  function which may return null (in which case, code should check it is not
  null), or a global, parameter or structure field declared with the null
  qualifier. (Use -nullderef to inhibit warning)
   q2.c(6,18): Storage ptr may become null
q2.c(8,2): Fresh storage ptr not released before return
  A memory leak has been detected. Storage allocated locally is not released
  before the last reference to it is lost. (Use -mustfreefresh to inhibit
  warning)
   q2.c(6,37): Fresh storage ptr created

Finished checking --- 2 code warnings
PS C:\Users\Himani\Desktop\SE\Assignment 3> 
```

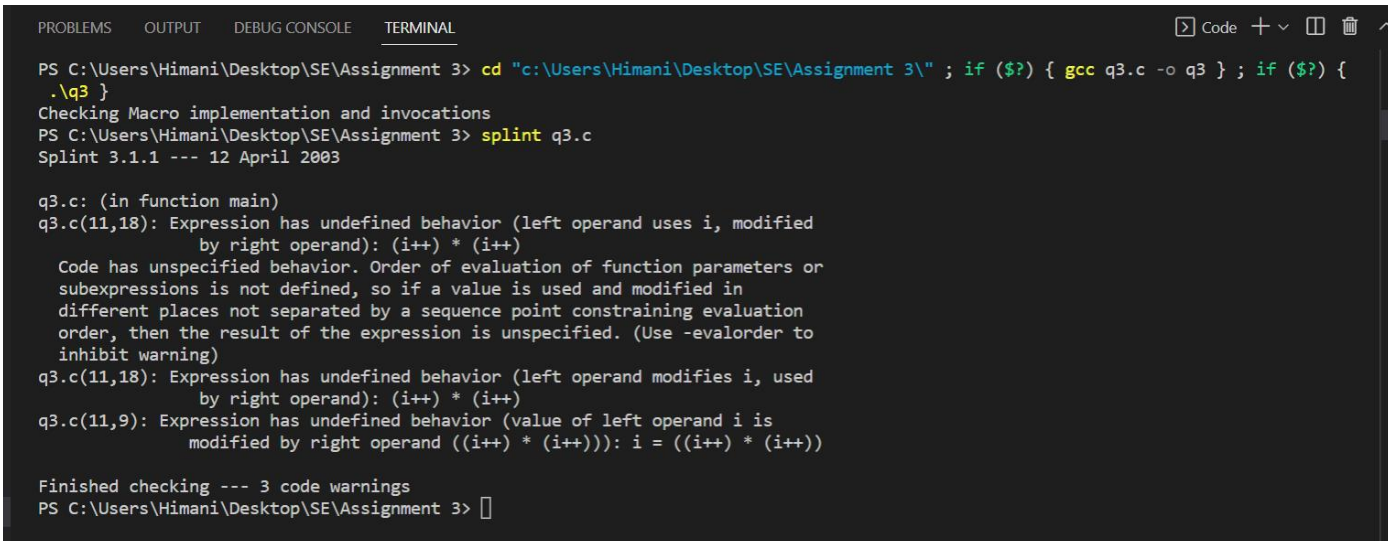
3. Macro implementations or invocations can be dangerous. Justify this statement by giving an example in C language.

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:

```c
#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:



```
PS C:\Users\Himani\Desktop\SE\Assignment 3> cd "c:\Users\Himani\Desktop\SE\Assignment 3\" ; if ($?) { gcc q3.c -o q3 } ; if ($?) {
 .\q3 }
Checking Macro implementation and invocations
PS C:\Users\Himani\Desktop\SE\Assignment 3> splint q3.c
Splint 3.1.1 --- 12 April 2003

q3.c: (in function main)
q3.c(11,18): Expression has undefined behavior (left operand uses i, modified
                by right operand): (i++) * (i++)
  Code has unspecified behavior. Order of evaluation of function parameters or
  subexpressions is not defined, so if a value is used and modified in
  different places not separated by a sequence point constraining evaluation
  order, then the result of the expression is unspecified. (Use -evalorder to
  inhibit warning)
q3.c(11,18): Expression has undefined behavior (left operand modifies i, used
                by right operand): (i++) * (i++)
q3.c(11,9): Expression has undefined behavior (value of left operand i is
               modified by right operand ((i++) * (i++))): i = ((i++) * (i++))

Finished checking --- 3 code warnings
PS C:\Users\Himani\Desktop\SE\Assignment 3> 
```

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.

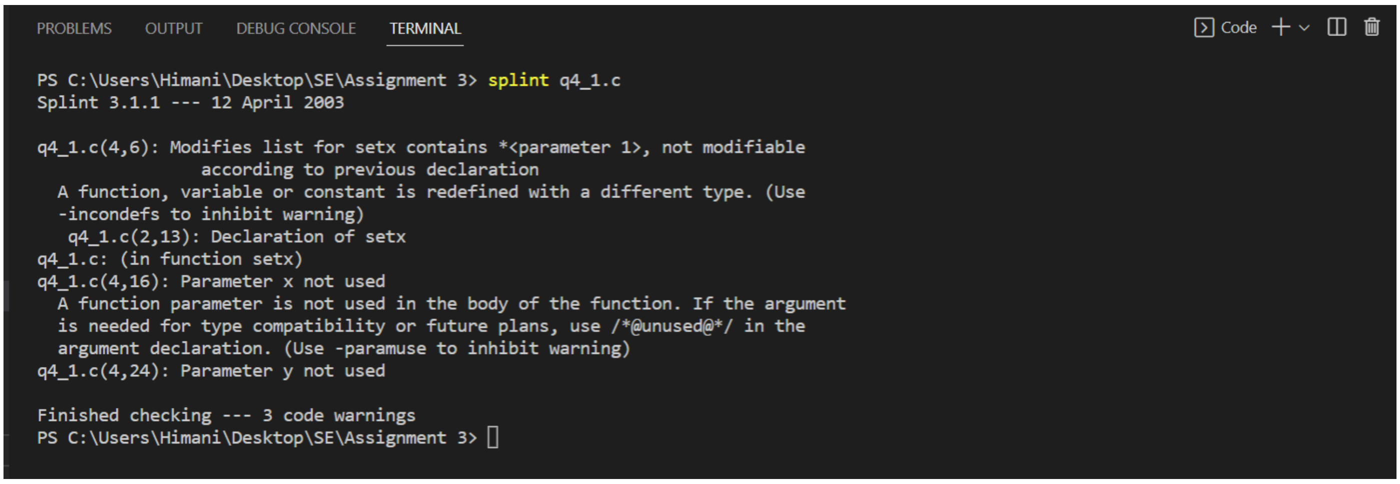
Functions communicate with their calling environment through an interface. The caller communicates the values of actual parameters and global variables to the function, and the function communicates to the caller through the return value, global variables and storage reachable from the actual parameters. 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:



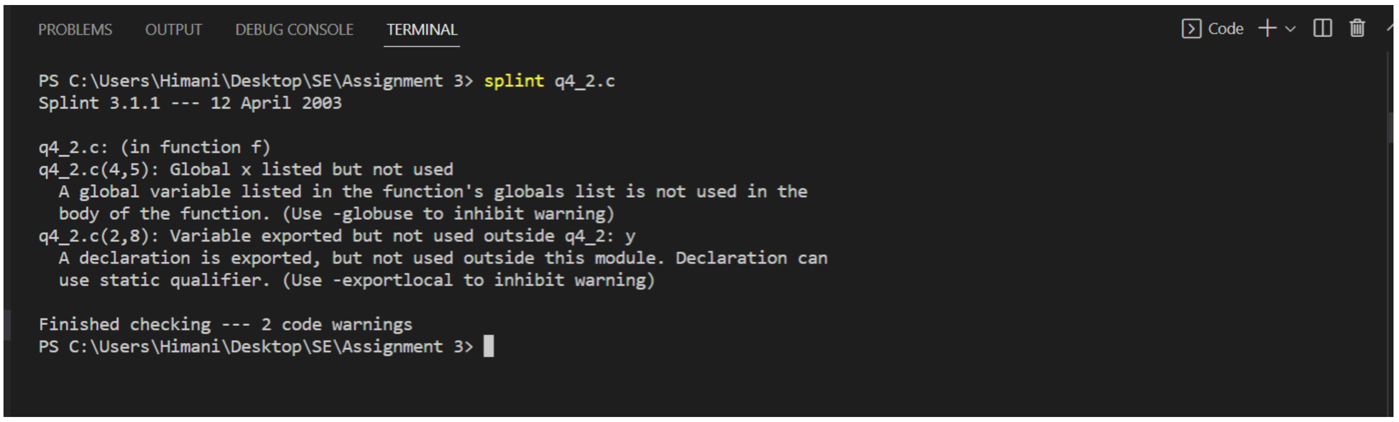
```
PS C:\Users\Himani\Desktop\SE\Assignment 3> splint q4_1.c
Splint 3.1.1 --- 12 April 2003

q4_1.c(4,6): Modifies list for setx contains *<parameter 1>, not modifiable
                according to previous declaration
  A function, variable or constant is redefined with a different type. (Use
  -incondefs to inhibit warning)
   q4_1.c(2,13): Declaration of setx
q4_1.c: (in function setx)
q4_1.c(4,16): Parameter x not used
  A function parameter is not used in the body of the function. If the argument
  is needed for type compatibility or future plans, use /*@unused@*/ in the
  argument declaration. (Use -paramuse to inhibit warning)
q4_1.c(4,24): Parameter y not used

Finished checking --- 3 code warnings
PS C:\Users\Himani\Desktop\SE\Assignment 3>
```

Global:

Source Code:

```
//Global
int x, y;

int f(void) /*@globals x;@*/ {
  return y;
}
```

Output:



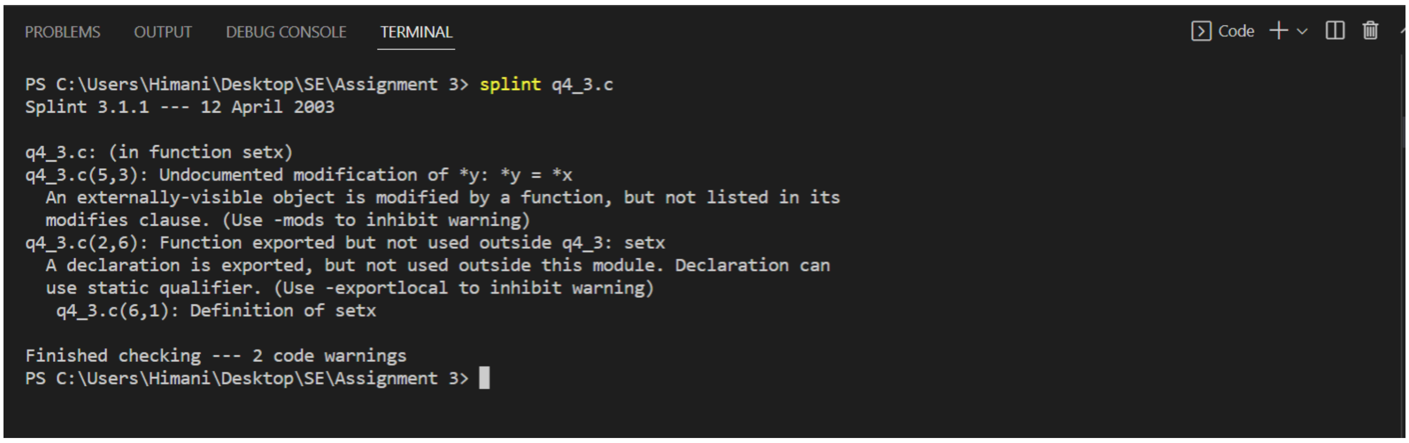
```
PS C:\Users\Himani\Desktop\SE\Assignment 3> splint q4_2.c
Splint 3.1.1 --- 12 April 2003

q4_2.c: (in function f)
q4_2.c(4,5): Global x listed but not used
  A global variable listed in the function's globals list is not used in the
  body of the function. (Use -globuse to inhibit warning)
q4_2.c(2,8): Variable exported but not used outside q4_2: y
  A declaration is exported, but not used outside this module. Declaration can
  use static qualifier. (Use -exportlocal to inhibit warning)

Finished checking --- 2 code warnings
PS C:\Users\Himani\Desktop\SE\Assignment 3>
```

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:



```
PS C:\Users\Himani\Desktop\SE\Assignment 3> splint q4_3.c
Splint 3.1.1 --- 12 April 2003

q4_3.c: (in function setx)
q4_3.c(5,3): Undocumented modification of *y: *y = *x
  An externally-visible object is modified by a function, but not listed in its
  modifies clause. (Use -mods to inhibit warning)
q4_3.c(2,6): Function exported but not used outside q4_3: setx
  A declaration is exported, but not used outside this module. Declaration can
  use static qualifier. (Use -exportlocal to inhibit warning)
   q4_3.c(6,1): Definition of setx

Finished checking --- 2 code warnings
PS C:\Users\Himani\Desktop\SE\Assignment 3>
```