Software Engineering (CS401)

Lab Assignment 3

**U19CS012**

Q1.) Implement the following **problematic control structures** in C and compare the outputs of **standard C compiler** and the **Splint tool**.

(A) Likely infinite loops

**Code**

*#include* <stdio.h>

*// Likely infinite loops*

int main()

{

    int x = 1;

*while* (x != 0)

    {

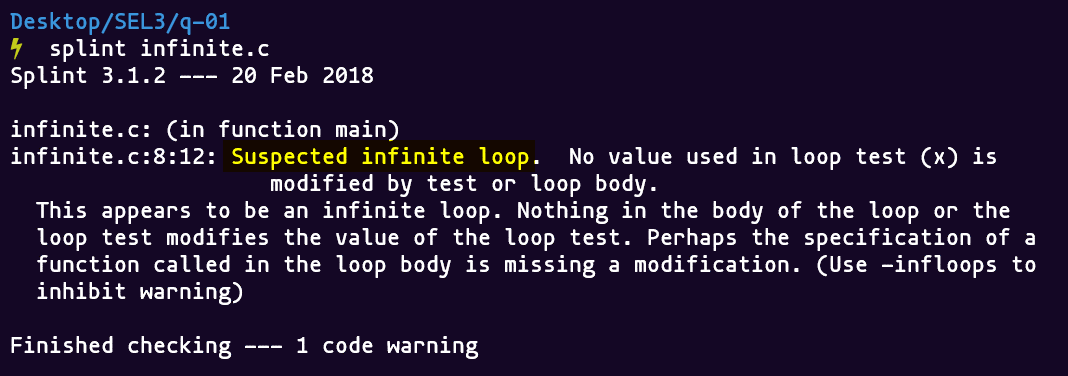
        printf(" %d", x);

    }

*return* 0;

}

**Output**



(B) Fall through switch cases

**Code**

*#include* <stdio.h>

*// Fall through switch cases*

int main()

{

    int x = 1;

*switch* (x)

    {

*case* 2:

        printf("2");

*case* 3:

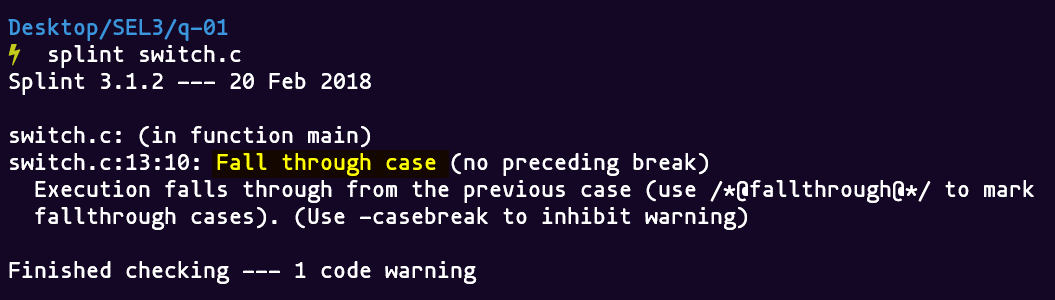
        printf("2");

    }

*return* 0;

}

**Output**



(C) Missing switch cases

**Code**

*#include* <stdio.h>

*// Missing switch cases*

typedef enum

{

    RED,

    YELLOW,

    GREEN,

} color;

int main()

{

    color x;

*switch* (x)

    {

*case* RED:

*break*;

*case* YELLOW:

        printf("No!");

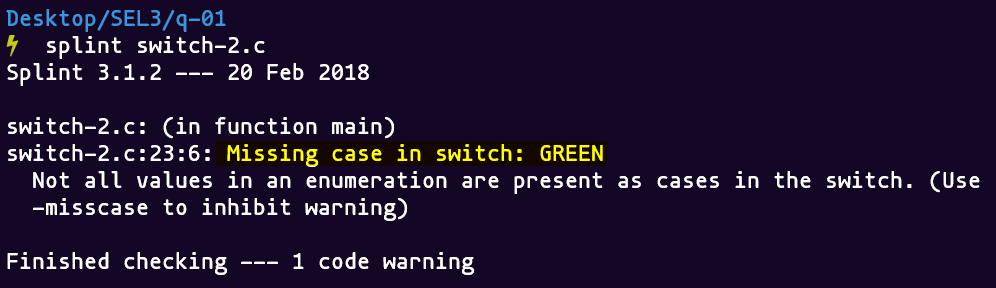
*break*;

    }

*return* 0;

}

**Output**



(D) Empty statement after an if, while or for

**Code**

*#include* <stdio.h>

*// Empty statement after an if, while or for*

int main()

{

    int x = 1;

*if* (x != 0)

        ;

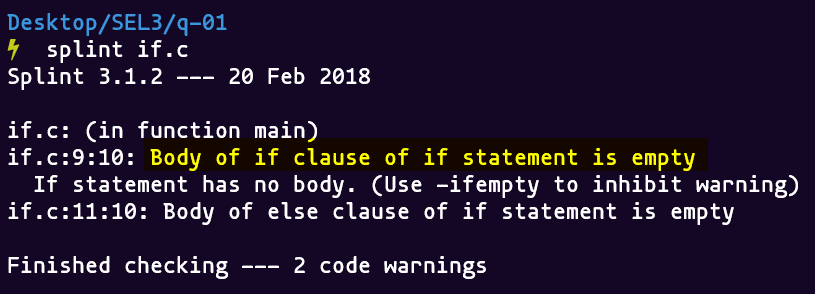
*else*

        ;

*return* 0;

}

**Output**



Q2.) What is **buffer overflow**? How it can be **exploited**? Write a C program to illustrate a buffer overflow attack?

A buffer overflow is basically when a **crafted section (or buffer)** of memory is written **outside of its intended bounds**. If an attacker can manage to make this happen from outside of a program it can cause security problems as it could potentially allow them to manipulate arbitrary memory locations, although many modern operating systems protect against the worst cases of this.

While both reading and writing outside of the intended bounds are generally considered a bad idea, the term "buffer overflow" is generally reserved for **writing outside the bounds**, as this can cause an attacker to easily modify the way your code runs.

**Code**

*#include* <stdio.h>

*#include* <string.h>

*// Example of Buffer OverFlow*

int main(int *argc*, char const \**argv*[])

{

    char str[4];

*// write past end of buffer (buffer overflow)*

    strcpy(str, "a string longer than 4 characters");

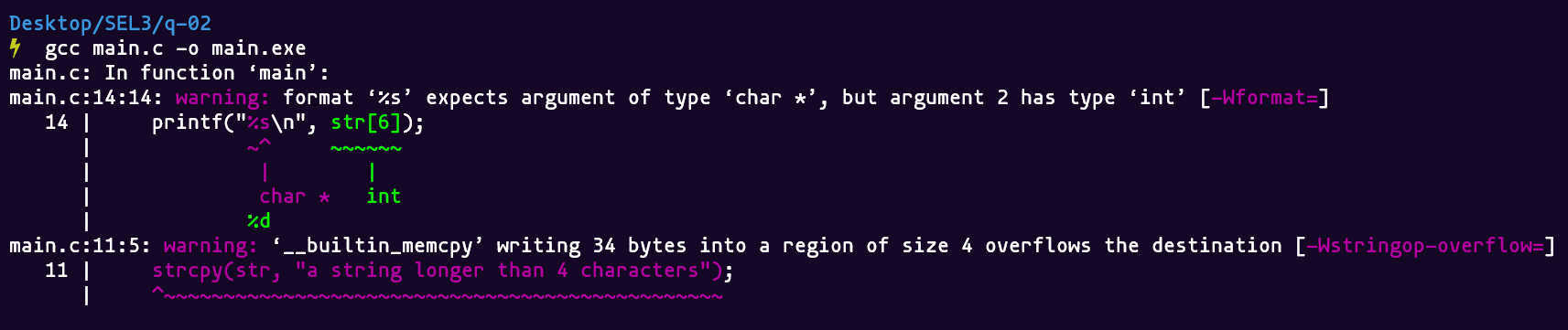
*// read past end of buffer (also not a good idea)*

    printf("%s\n", str[6]);

*return* 0;

}

**Output**



Q3.) “*Macro implementations or invocations can be dangerous*.” Justify this statement by giving an example in C language.

An **unsafe function-like** macro is one that, when expanded, evaluates its argument more than once or does not evaluate it at all. Contrasted with **function calls**, which always evaluate each of their **arguments exactly once**, unsafe function-like macros often have unexpected and surprising effects and lead to subtle, hard-to-find defects.

Consequently, every function-like macro should evaluate each of its arguments **exactly once**. Alternatively and preferably, defining function-like macros should be avoided in favor of inline functions.

**Code**

*#include* <stdio.h>

*#define* abs(*i*) ((i) >= 0 ? (i) : -(i))

int main()

{

    int x = -5;

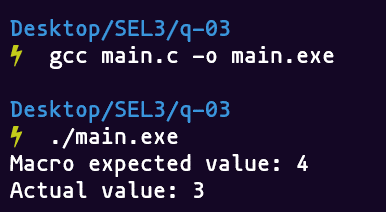
*// Should return 4 but returns 3*

    printf("Macro expected value: 3\nActual value: %d\n", abs(++x));

*return* 0;

}

**Output**



Q4.) 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.

(A) Modification

**Code**

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

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

{

    \*y = \*x;

}

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

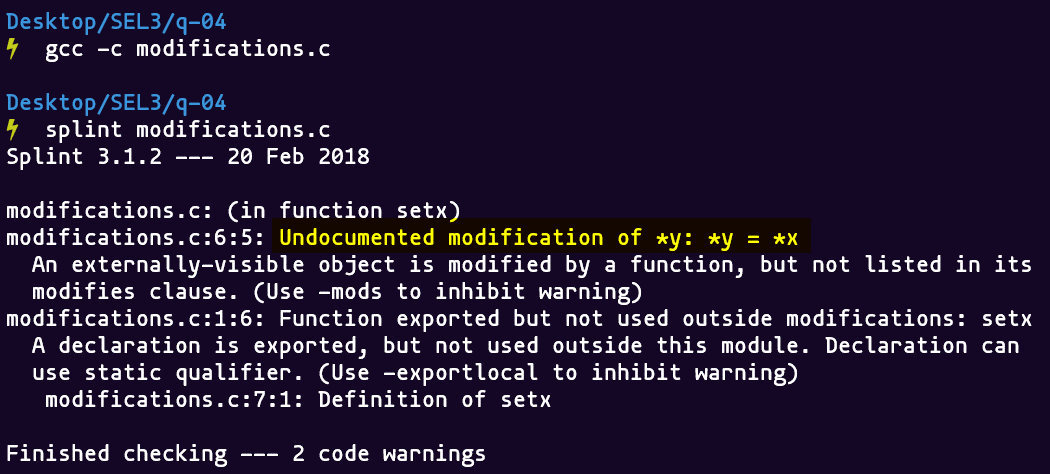
*/\*@modifies \*y@\*/*

{

    setx(y, x);

}

**Output**



(B) Accessing Global Variables

**Code**

int x, y;

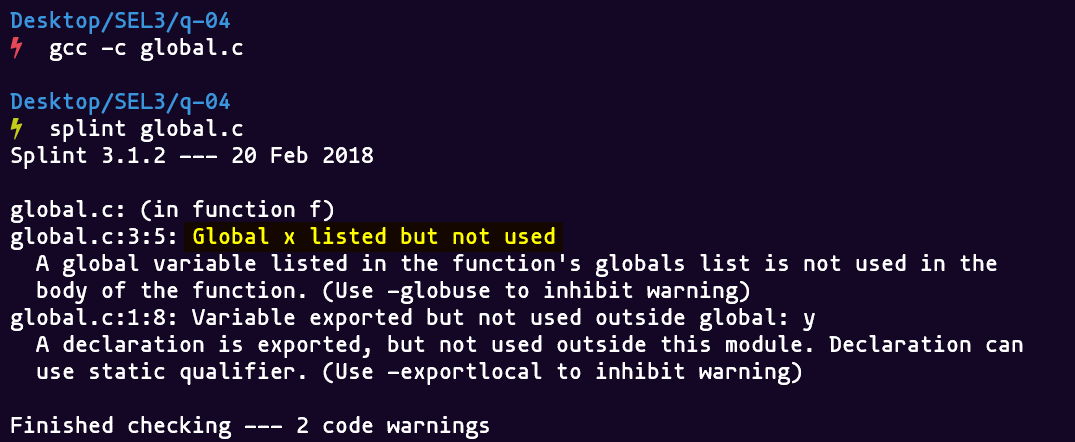
int f(void) */\*@globals x;@\*/*

{

*return* y;

}

**Output**



(C) Declaration Consistency

**Code**

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

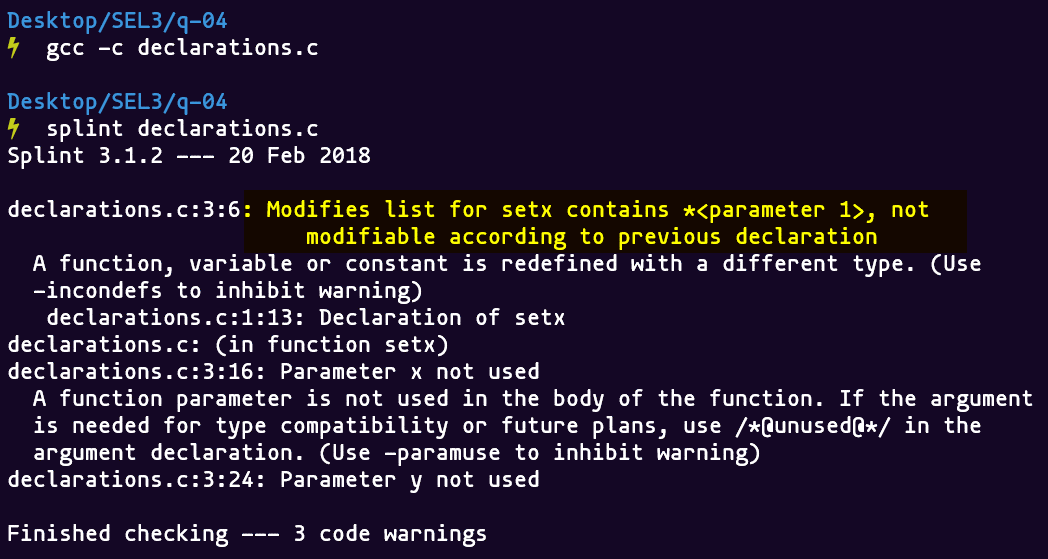
void setx(int \**x*, int \**y*) */\*@modifies \*x@\*/*

{

*// do stuff*

}

**Output**



**SUBMITTED BY**: U19CS012

BHAGYA VINOD RANA