Software Engineering (CS401)

Lab Assignment 3

U19CS012

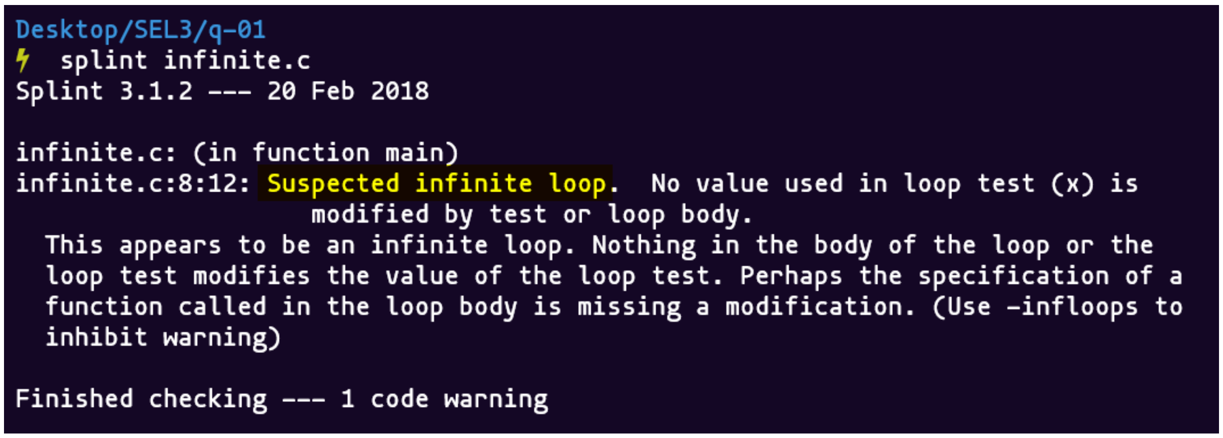
Q1.) Implement the following **problematic control structures** in C and compare the outputs of **standard C compiler** and the **Splint tool**.

(A) Likely infinite loops

Code

```c
#include <stdio.h>

// Likely infinite loops

int main()
{
    int x = 1;
    while (x != 0)
    {
        printf(" %d", x);
    }
    return 0;
}
```

Output



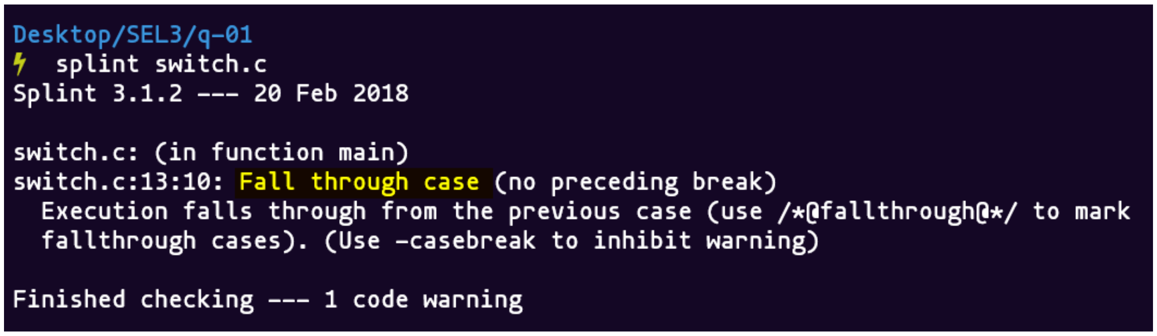
```
Desktop/SEL3/q-01
ϟ  splint infinite.c
Splint 3.1.2 --- 20 Feb 2018

infinite.c: (in function main)
infinite.c:8:12: Suspected infinite loop.  No value used in loop test (x) 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
```

(B) Fall through switch cases

Code

```c
#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



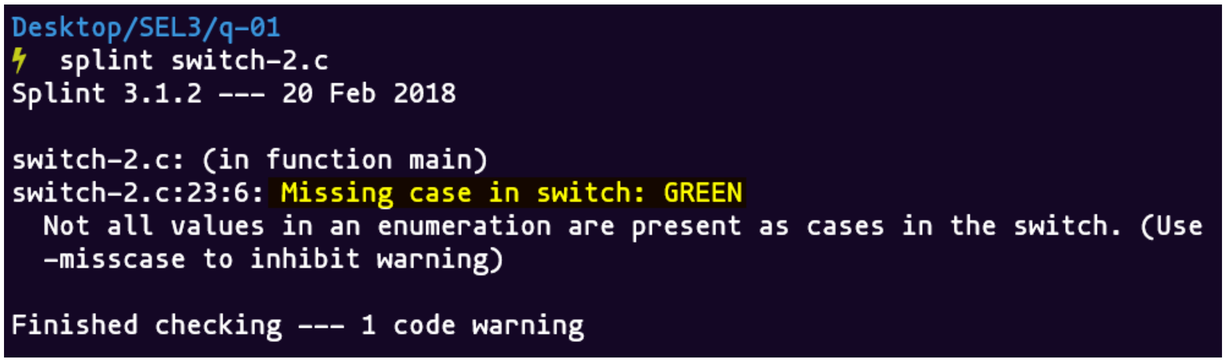
```
Desktop/SEL3/q-01
ϟ  splint switch.c
Splint 3.1.2 --- 20 Feb 2018

switch.c: (in function main)
switch.c:13:10: Fall through case (no preceding break)
  Execution falls through from the previous case (use /*@fallthrough@*/ to mark
  fallthrough cases). (Use -casebreak to inhibit warning)

Finished checking --- 1 code warning
```

(C) Missing switch cases

Code

```c
#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



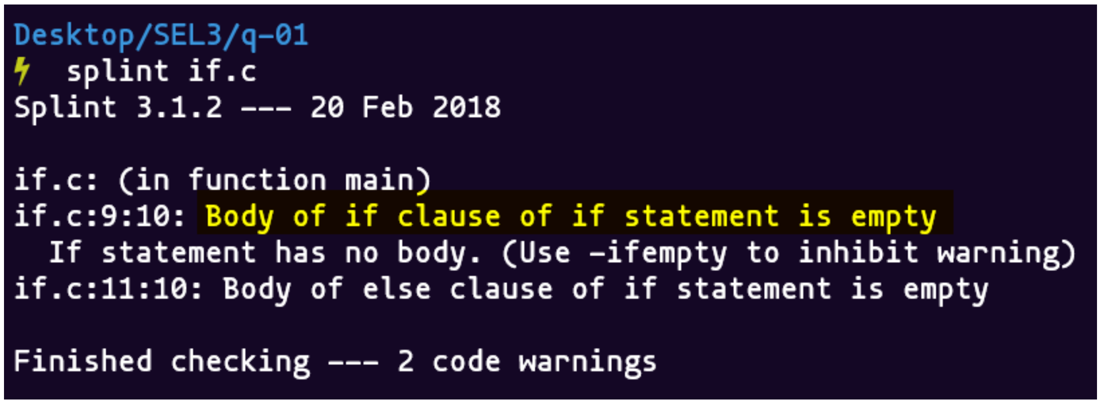
```
Desktop/SEL3/q-01
ϟ  splint switch-2.c
Splint 3.1.2 --- 20 Feb 2018

switch-2.c: (in function main)
switch-2.c:23:6: Missing case in switch: GREEN
  Not all values in an enumeration are present as cases in the switch. (Use
  -misscase to inhibit warning)

Finished checking --- 1 code warning
```

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

Code

```c
#include <stdio.h>

// Empty statement after an if, while or for

int main()
{
    int x = 1;
    if (x != 0)
        ;
    else
        ;

    return 0;
}
```

Output



```
Desktop/SEL3/q-01
  splint if.c
Splint 3.1.2 --- 20 Feb 2018

if.c: (in function main)
if.c:9:10: Body of if clause of if statement is empty
  If statement has no body. (Use -ifempty to inhibit warning)
if.c:11:10: Body of else clause of if statement is empty

Finished checking --- 2 code warnings
```

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

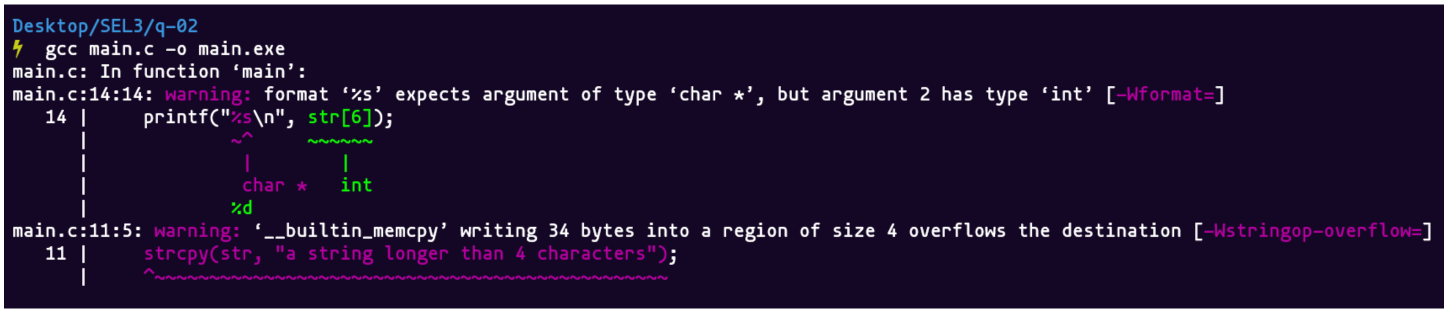
A buffer overflow is basically when a **crafted section (or buffer)** of memory is written **outside of its intended bounds**. If an attacker can manage to make this happen from outside of a program it can cause security problems as it could potentially allow them to manipulate arbitrary memory locations,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

```c
#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



```
Desktop/SEL3/q-02
ϟ  gcc main.c -o main.exe
main.c: In function ‘main’:
main.c:14:14: warning: format ‘%s’ expects argument of type ‘char *’, but argument 2 has type ‘int’ [-Wformat=]
   14 |     printf("%s\n", str[6]);
      |             ~^     ~~~~~~
      |              |     |
      |              char *   int
      |             %d
main.c:11:5: warning: ‘__builtin_memcpy’ writing 34 bytes into a region of size 4 overflows the destination [-Wstringop-overflow=]
   11 |     strcpy(str, "a string longer than 4 characters");
      |     ^~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

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

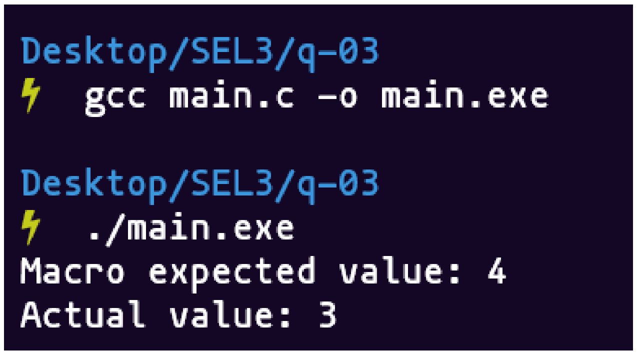
An **unsafe function-like** macro is one that, when expanded, evaluates its argument more than once or does not evaluate it at all. Contrasted with **function calls**, which always evaluate each of their **arguments exactly once**, unsafe function-like macros often have unexpected and surprising effects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

```c
#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



```
Desktop/SEL3/q-03
ϟ  gcc main.c -o main.exe

Desktop/SEL3/q-03
ϟ  ./main.exe
Macro expected value: 4
Actual value: 3
```

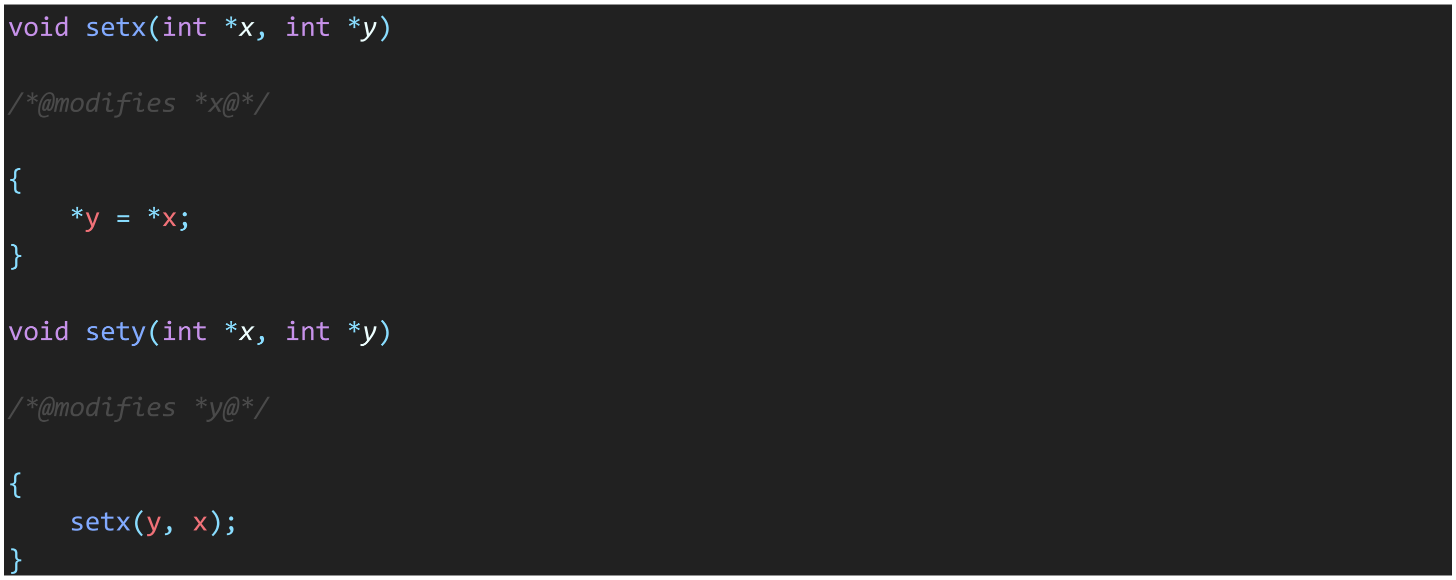
Q4.) What do you mean by **Interface faults**. Write a set of C programs to implement interface faults and perform their detection using Splint tool. Check whether they are detected by the **standard C compiler** or not.

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.

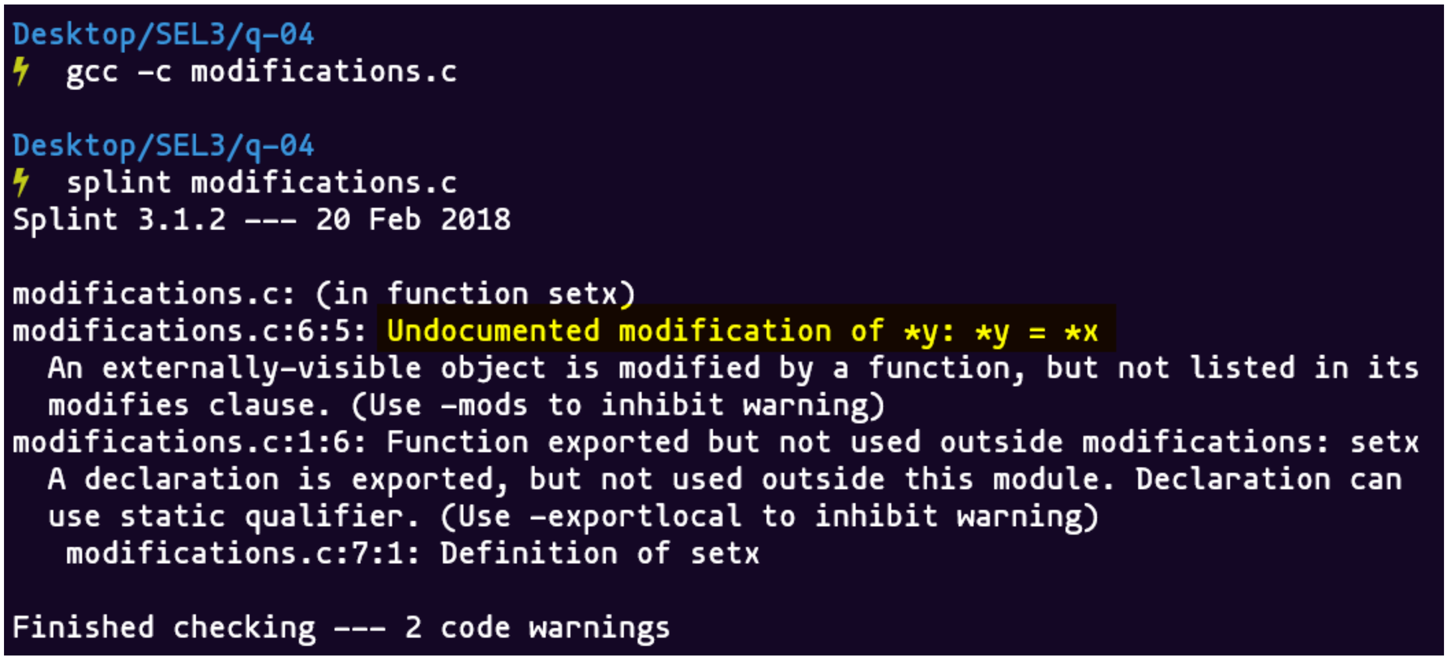
(A) Modification

Code



```c
void setx(int *x, int *y)

/*@modifies *x@*/

{
    *y = *x;
}

void sety(int *x, int *y)

/*@modifies *y@*/

{
    setx(y, x);
}
```

Output



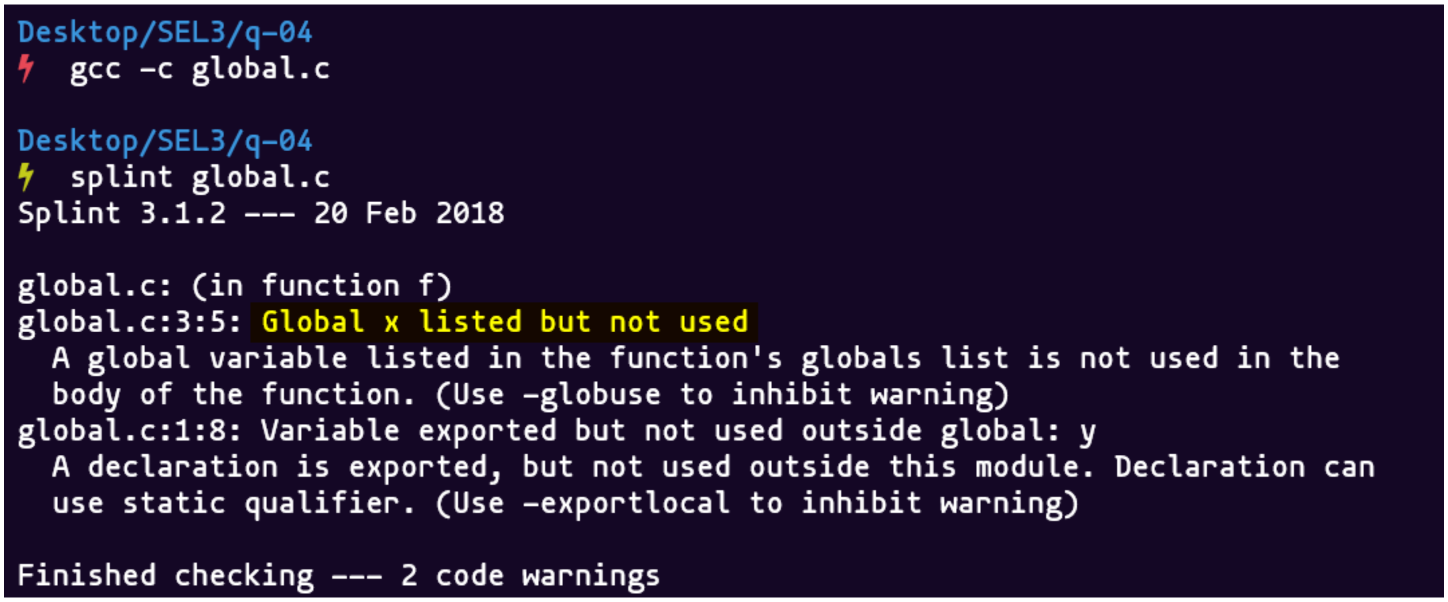
```
Desktop/SEL3/q-04
ϟ  gcc -c modifications.c

Desktop/SEL3/q-04
ϟ  splint modifications.c
Splint 3.1.2 --- 20 Feb 2018

modifications.c: (in function setx)
modifications.c:6:5: 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)
modifications.c:1:6: Function exported but not used outside modifications: setx
  A declaration is exported, but not used outside this module. Declaration can
  use static qualifier. (Use -exportlocal to inhibit warning)
   modifications.c:7:1: Definition of setx

Finished checking --- 2 code warnings
```

(B) Accessing Global Variables

Code

```c
int x, y;

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

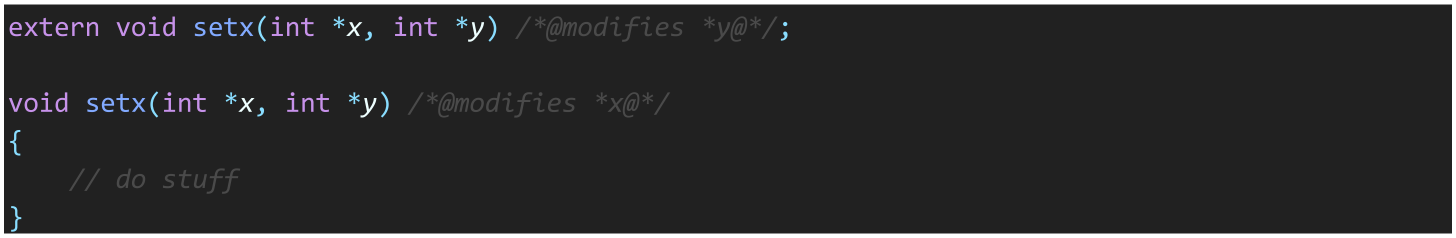
Output



```
Desktop/SEL3/q-04
ϟ  gcc -c global.c

Desktop/SEL3/q-04
ϟ  splint global.c
Splint 3.1.2 --- 20 Feb 2018

global.c: (in function f)
global.c:3:5: 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)
global.c:1:8: Variable exported but not used outside global: 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
```

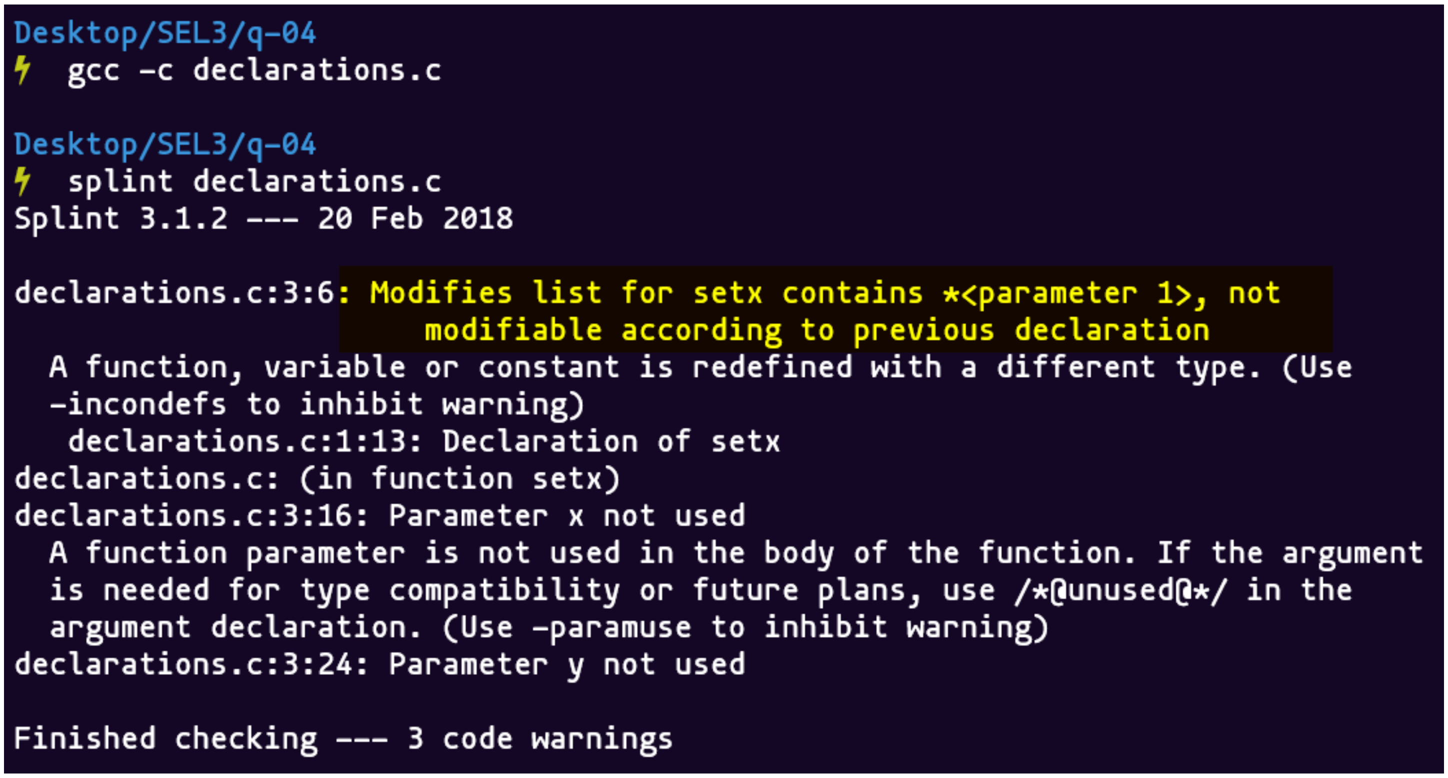
(C) Declaration Consistency

Code



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

void setx(int *x, int *y) /*@modifies *x@*/
{
    // do stuff
}
```

Output



```
Desktop/SEL3/q-04
ϟ  gcc -c declarations.c

Desktop/SEL3/q-04
ϟ  splint declarations.c
Splint 3.1.2 --- 20 Feb 2018

declarations.c:3:6: 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)
   declarations.c:1:13: Declaration of setx
declarations.c: (in function setx)
declarations.c:3:16: 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)
declarations.c:3:24: Parameter y not used

Finished checking --- 3 code warnings
```

SUBMITTED BY: U19CS012

BHAGYA VINOD RANA