SOFTWARE ENGINEERING

Assignment-1

SUHAS JAIN

19CS30048

1) Problem:

The bug in this code lies in the line

```
if(a==sqr(b)).
```

This error is due to internal precision errors while rounding up floating point numbers.

Solution:

The way we can fix this by using a tolerance number and making sure the difference between a and sqr(b) is within that tolerance rather than directly equating them. So something like:-

```
if (abs(a - sqr(b)) < 1e-6)
```

Here 10^{-6} is the tolerance number. We can change this according to problem. statement we have.

Guideline

We should never compare 2 floating point numbers using the == operator because this does a bit - by - bit check. So, due to interal precision errors, we are very likely to get mistakes in our codes like this. These mistakes are also very hard to debugg. Instead we should check the absolute difference between them. And then check if it is less than a permissible tolerance value.

SUHAS JAIN 19CS30048

2) Problem

When we run the code we get a runtime error which says `Segmentation fault (Core dumped)`. This is obviously not what the developers intended. I think the developer wanted to check if the memory has been allocated the value '5' or not using the new operator.

After using `int* p = new int(5)` ie. a value of 5 is stored in a memory location, and p is an integer pointer pointing to that location.

The problem lies in the line `if(p=0)`. This is an assignment operator rather than a equality operator. In this statement p gets assigned 0 and the term evaluates as false. So when we then want to print *p, we're toying to dereference a NULL pointer, hence the segmentation fault.

Solution

If we change our if condition to either ~~if~~ `if (p==0)` or `if (p==NULL)` this, we will make our code work as developer intended.

Guideline

We should have a clear distinction in our mind between the assignment `operator (=)` and the rational `operator (==)` and we should use them in appropriate places.

SUHAS JAIN 19CS30048

We should also check if the pointer is NULL or not before dereferencing it. If we de-reference a NULL pointer we will get a seg-fault like we got in the original code.

3) Case 1: Line 2 and 3 are commented out.

In the unoptimised build, there is no optimisations enabled. So each line is executed steps-by-step. So in the first condition $r == 0$ get evaluated as true so the compilor does not check the 2nd condition. In the 2nd if condition rem(n, r) is called first so $n \% r$ (where r is 0) results in a floating point exception.

In the optimised build, there are many optimisations compilor does. There is an optimisation called "constant folding". In this process the compibr takes the expressions whose values can be calculated directly at the compile time and replaces them with the calculated values directly. Also compiler identifies duplicate expressions directly and they are rewritten with the same result.

So in this code values of n and r are available at the compile time. Compilor identifies both of these statements as identical and replaces both of them with true because $r == 0$ is evaluated as true.

Case 2: Line 1 is commented out, 2 & 3 are present

The difference from case 1 is that values of r and s are unknown at compile time so optimisations cannot take place as compiler cannot replace both if statements as true even when it knows both if are equivalent.
The expressions are evaluated at runtime in both optimised and unoptimised builds. So when evaluation of 2nd if statement takes place it throws an exception.

Guideline

While executing the code in a release (optimised) build, the compiler applies certain optimisations. The level of these optimisations can also be changed as O1 O2 O3 etc.
But in case we have only one if statement like if (r==0 || gcm(n,r)) this will cause no problem but it is still a bad practice since it's our duty to check that there is no exception in any of the statements in if loop.
It is always advisable to build a program in a debug build before release build. In the debug build, we will get to know if there is any bug or exception which may get shipped in the release build.

SUHAS JAIN 19CS30048

4)

Function Name	Behaviour	Justification & Comments
$f_1()$	char* str = "Bat" Segmentation fault (core dumped) First line of output is correctly printed after that runtime exception occurs.	"Bat" is treated as a constant string (rvalue) during the assignment, but LHS which is a Lvalue is not constant. When we try to change str[0] to 'C', we get a runTime error.
$f_2()$	str[0] = 'C' throws a Compilation error	Here, since we have made the RHS const in this case we don't have a problem in the first statement. str is a char pointer pointing to a constant string. That's why str[0] cannot be reassigned to 'C'. That's why we get a compilation error.
$f_3()$	~~str[0]~~ str = "Rat" throws a Compilation error	When we write char* const str = "Bat", here str is a constant pointer, pointing to the string "Bat". So, trying to make it point to a new string "Rat" results in compilation error.

$f_4()$	Bat Cat Rat Here we get the correct and expected output.	The function ~~sto~~ strdup creates a copy of the ~~strip~~ string passed to it & returns a pointer to the same. So here neither the pointer nor the string it is pointing to is constant. So it can be modified in whatever way we want to.
$f_5()$	str[0] = 'c' gives Compilation error.	Here we made the string constant in the first line when equated to "Bat". So we cannot change the string str[0] = 'c' gives a compilation error.
$f_6()$	str = "Rat" gives Compilation error.	Here the first line we make the pointer of the string is made constant so it cannot point to any new string. when we equate it to "Rat" we get compile error.

SUHAS JAIN 19CS30048

Guidelines :-

- Any constant values on the right side (here strings) should also be stored in a const variable on the left side, so as to maintain uniformity.
- Character strings are by default treated as constants. That is why one ~~cant~~ can use the strdup method to make a copy of a string. Because it creates a copy and assigns so we need not make pointer or string const on LHS.
- One should understand when which ~~id~~ item is constant. For example const char* str makes the strings pointed to be str as constant. char * const str makes the pointer itself as a constant pointer so it ~~cannot~~ cannot point to any new location in the future.

PTO..

5)

SUHAS JAIN 19CS30048

Line No.	Behaviour	Justification and comments.
1	Compilation Error	The LHS is a non-const lvalue reference of type 'int&' and RHS is a rvalue of type 'int'. 'int&' cannot be assigned to a constant literal because these cannot be modified.
2	Compilation Error	Again similar to Line 1 we are trying to bind a non-const lvalue reference of type 'int&' to an rvalue of type 'int.' which gives compilation error. Both e and f return a integer literal.
3	We get an output as x=10 (same as a) But address of x is different from that of a.	In g copy of the variable a is passed. Hence the value of both are same but both are stored in a different address. Also variable x gets destroyed after the function call ends. so that is why we get a warning as it is not advised to pass reference of a local variable.
4	We get a correct expected output. The values of a and x are same (=10) and their address are also the same.	In function h a is passed by reference and then returned by reference so a and rrr basically point to same memory location.

SUHAS JAIN 19CS30048

5	Compilation error at Line 1 so this has no ~~meaning~~ meaning.	Line 5 uses variable of line 1. and line 1 itself throws a compilation error so this has no meaning.
6	Due to compilation error in line 2 ~~this~~ line 6 has no meaning.	Variable does not get assigned any value in the line 2 so because of that error only this line has no meaning.
7	Segmentation fault (core dumped) which means a runtime error.	As we're returning the reference of a local variable and it gets destroyed so memory is not available, so hence, accessing it causes a segmentation fault.
8	Correct expected output. The values a, x, ~~xxx~~ are all same (=10) and their addresses are also all same.	In the function h we have passed and returned by reference so all variables point to same memory location and store the same value.
9	The value of a ~~a~~ and x printed are same but the address are different.	e(a) returns a constant local value. So, it cannot be bound to 'int&'. So we fix this by using a const reference. And it is passed by value x is a copy of ~~a~~ a. So values are same but ~~addr~~ addresses are different.

10	Printed values of a and x as well as the addresses are also same.	as a is passed by reference value and addresses are same. Issue of const RHS is solved by putting const keyword.
11	Values of a and x are same but their addresses are different.	Since it is pass by value. The address is different. But return by reference of a local variable is not safe.
12	The values of a and x are same and addres are also same.	Same as line 4.
13	Values of a, x, ~~xxx~~ xvvc are same but all address are different.	x is copy of a, ~~xx~~ and it is also return ~~by~~ value so xvvc is also a copy. Therefore all different ~~variables~~. addresses.
14	Values of a, x, xvvc are same address of a and x are same but diff for xvvc.	x is due to pass by reference xvvc is a copy.

15	Segmentation fault.	Here since the function of returns value of local variable Error.
16	Correct output- All same values and same address	All variables point to same memory location.
17	Compilation Error.	There should be a lvalue on the RHS but here we have a constant literal
18	18 is dependent on 17.	Same as line 17
19	Compilation Error.	Similar to line 17. We cannot have a lvalue on RHS side.

15	Segmentation fault.	Here since the function returns value of local variable Error.
16	Correct output- All same values and same address	All variables point to same memory location.
17	Compilation Error.	There should be a lvalue on the RHS but here we have a constant literal
18	18 is dependent on 17.	Same as line 17
19	Compilation Error.	Similar to line 17. We cannot have a lvalue on RHS side.

SUHAS JAIN 19CS30048

20	Dependent on line 19	
21	Segmentation fault in the end (runtime error)	We cannot assign 3 to reference of a local variable. The memory gets destroyed after function call.
22	Dependent on line 21 Control does not reach here	Similar to 18 & 20.
23	Function prints value and address of a. 4 gets assigned to a	Since all point to same memory location. all a are assigned 4.
24.	Value of a is printed as 4. Same address as before	Same as line 23.

SUHAS JAIN 19CS30042

Guidelines :-

- Never return a reference to a local variable from a function as that memory is unallocated after we return from that function.
- While assigning a constant or temporary value to a reference, keep the lvalue a const reference.
- Keep in mind the fact that there should be lvalue to the right of the assignment operator. There should not be any constant or temporary returned value in the left of an assignment operator.
- Also, be cautious while passing a reference to a variable as any change done there will also be reflected in the original variable.