Kaushal Banthia
19CS10039

① The bug is due to inaccuracies when handling fractional numbers. a and sqr(b) don't exactly. So either we can round them off both to certain decimal points, or we can ~~the~~ take the absolute value of the difference between a and sqr(b) and ensure that the value is lesser than a certain small value (say 10^{-8}).

⇒ To prevent this bug, we can avoid using the equality test, and instead use $|a - sqr(b)| <$ some_small_value (say 10^{-8}).

② ~~This is~~ The output of the program is nothing. (An error).
This is not what the developer intended.
The bug is the assignment operator that is used inside the condition for the if statement. It should be replaced by the == operator.

⇒ Such bugs can be avoided by carefully looking out for the placement of '=' and '=='. '=' is used for assigning values to variables, whereas '==' is used for comparing values. Also, use of NULL pointer is preferred.

③ In the first case, for the debug (un-optimized) build, the compiler ~~executes~~ compiles the code as it is, without removing anything, or making any replacements.
For the release (optimized) build, the compiler replaces the condition (r == 0 || rem(n,r)) with true, since r is assigned the value of 0, at the beginning of the program. This makes the first condition of the if statement to be true. Hence, since the condition has an OR, the compiler replaces the condition with true, since one condition is true always, (r == 0). This causes the compiler to never encounter the condition (rem(n,r)).

In the debug (un-optimized) build, no replacement is made and hence, the condition (rem(n,r)) is encountered, which causes the division by 0 error.

Now, when we comment line 1 and uncomment lines 2 and 3, we are not allowing the compiler to replace the condition inside the if statement with true, in the release (optimized) build. This is because, now the condition r == 0, is not always true. It is now dependent on the input value of r.

Hence, the compiler cannot do the replacement.

Kaushal Banthia
19CS10039

In the debug (un-optimized) build, no change is observed, since the compiler does not do ~~the~~ the replacement anyways.

The main point is that, now the value of r is dependent on user input and hence the compiler cannot take it for granted that the input will always be '0', even if we always enter the ~~value~~ value '0'.

⇒ To avoid such bugs, we can always take inputs of such variables from the user, instead of ~~defining~~ initialising them ~~befor~~ beforehand with our own values.
We should only ~~define~~ initialise those variables, that are not used in comparisons.

4

Function Name	Behaviour	Justification & Comments.
f1()	Run-Time Exception	This exception happens due to the line `str[0]='c';` This is due to the fact that char * is a constant character array in C++. Thus, when the code is executed and the above line tries to change the variable's character, a runtime exception occurs.
f2()	Compilation Error	This error happens due to the line `str[0]='c';` This is due to the fact that the variable is defined as a const char * (pointer to const char). Since it is const, it is not possible to modify the contents ~~of~~ that the variable points to.
~~f3~~ f3()	Compilation Error	This error happens due to the line `str = "Rat";` Here, the problem is that the variable is a constant pointer and wes are changing its value, which is not possible.
f4()	Correct Output - Showing the Output	Here the pointer and the char are both non-const. Thus, both of them can be modified without causing any errors.

Kaushal Banthia
19CS10039

3

Function Name	Behaviour	Justification & comments.
f5()	Compilation Error	This error ~~due~~ happens due to the line str[0] = 'c'. This is due to the fact that the variable declared is a pointer to a const char. Hence, the pointed string cannot have its characters mutated.
f6()	Compilation Error	This error is due to the line str = strdup ("Rat"). This is due to the fact that the variable declared is a constant pointer to a char, Thus, the modification of the pointer (a const) is not possible.

④ Some guidelines are

⇒ Use of char * should be avoided, since C++ has provided the new datatype of string. this is a much more efficient way to contain the variables that would previously have required char[] or char*.

⇒ Now, even though string is provided, if we want to use char *, then using a ~~const and char~~ pointer to a constant char is the best choice, ie, const char *.

⇒ we should look at the placement of the word 'const', since it determines whether the pointer is a const or the variable pointed to, is a const.

⑤ Line	Behaviour	Justifications & comments.
1	Compilation Error.	The function returns a temporary const value. Thus, the reference to non-const cannot be initialised with this temporary value.
2	Compilation Error.	~~Reference~~ The function returns a non-const temporary value. Thus, the reference to non-const cannot be initialised with this temporary value

Kaushal Banthia
19CS10039

Line	Behaviour	Justification & Comments
3	Unpredictable Behaviour.	Reference to a local variable is returned, which is strictly not recommended, since the local variable may not exist anymore.
4	Correct Output - ~~Showing~~ Showing the Output.	Returns reference to the variable which was passed in the main function. Proper initialisation.
5	Compilation Error	Line 1 caused the error.
6	Compilation Error	Line 2 caused the error.
7	Unpredictable Behaviour.	Line 3 caused the error.
8	Correct Output - Showing the Output.	Line 4 was perfect and thus Line 8, that uses variable defined in Line 4 also works perfect.
9	Correct Output - Showing the Output.	The function returns a temporary value, which is fine to initialise a reference to a const.
10	Correct Output - Showing the Output.	The function returns a temporary value, which is fine to initialise a reference to a const.
11	Unpredictable Behaviour.	Reference to a local variable is returned, which may not even exist anymore.
12	Correct Output - Showing the Output.	Returns reference to the variable which was passed in the main function Proper initialisation.
13	Correct Output - Showing the Output.	Line 9 was perfect and thus Line 13 is perfect.
14	Correct Output - Showing the Output	Line 10 was perfect and thus line 14 is perfect.
15	Unpredictable Behaviour.	Line 11 caused the error.
16	Correct Output - Showing the Output.	Line 12 was perfect and thus line 16 is perfect.
17	Compilation Error.	The function is returning a temporary value, to which we can't assign yet another value.

Kaushal Banthia
19CS10039

5

Line	Behaviour	Justifications & Comments
18	~~Compilation Error~~ Output that is not expected	~~Line 17 caused the error.~~ The value of 'a' is unchanged. (Function call intended to cause some change
19	Compilation Error	The function is returning a temporary value, to which we can't assign yet another value.
20	~~Compilation Error~~ Output that is not expected.	The value of 'a' is unchanged. Function call intended to cause some change.
21	Unpredictable Behaviour	The reference to a local variable is ~~passed~~ (returned, which may not even exist. Assigning values to such references is wrong.
22	Output that is not expected	The value of 'a' is unchanged. ~~function~~ Function call intended to cause some change.
23	Correct Output - Showing the Output.	The value of 4 is assigned to the ~~reference that is returned~~ reference to 'a', that is returned by the function h(a).
24	Correct Output - Showing the Output	The correct (changed) value of 'a' is now displayed and the purpose of the function call (to cause some change) has been fulfilled.

Some guidelines are

⇒ Always use a const reference.

⇒ Never ever return reference to a local variable, as it may cease to exist.

⇒ Never try to assign a temporary return value to another value (say 4) and vice versa.
Only such assignment that can be made is for assigning a value (say 4) to a non-constant reference.

⇒ Always try to ensure that a function fulfills its purpose.