Ashutosh Kumar Singh 19CS30008 __/__/____

Software Engineering
Assignment 1

1. The bug is in the line :- `if (a == sqr(b))`.
A mismatch is printed due to internal precision errors while rounding up floating-point numbers.

The way to fix this is to use a very small number as a tolerance value, which means that we will compare these two numbers only upto a certain precision. So, if we take the tolerance value as 10^{-6} (can be anything depending on our requirement), the correct line of code will be:-

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

Guideline:-
We should never compare two floating-point numbers using the == operator, because this does a bit-by-bit check. So, due to internal precision errors, we are very likely to get errors.
Instead, we should check that the absolute difference between the two numbers is less than a permissible tolerance value or not.

Ashutosh Kumar Singh 19CS30008 __/__/____

2. The output of the given program is :-
[Segmentation Fault (core dumped)],
which is basically a runtime error.

No, this is surely not what the developer intended.
Probably the developer wanted to check whether the memory had been properly allocatted or not. If it had not been allocatted, he/she wanted to print "No value", else they wanted to print the value pointed to by the pointer p (which is 5 in this case).

So, the developer wrote :- int *p = new int(5), i.e., a value of 5 is stored in a memory location, and p is an integer pointer, pointing to this memory location.

After this, the developer wrote :- [if (p=0)]. This is what caused the bug. We know = is an assignment operator. So, p gets assigned 0 (or NULL reference), and then this condition evaluates to false, so the control reaches the else block. In the else block, we want to print *p. So, in this case, we are trying to de-reference a NULL pointer, which gives a Segmentation Fault.

The way to fix this is to change the if condition to either one of the following :-
[if (p==0)], or, [if (p==NULL)]

Ashutosh Kumar Singh 19CS30008 __/__/____

Guideline:-

We should be careful in the usage of the assignment operator(=) and the relational operator(==), and understand the differences between them.

Also, we should be careful while de-referencing a pointer, and should check if it is NULL or not, because trying to de-reference a NULL pointer may lead to a segmentation fault (runtime error).

3. When line 1 is present and Lines 2 & 3 are commented:-

In the debug (unoptimized) build, there are no optimizations enabled, so each line of the code is executed step-by-step. So, in the first if condition, r==0 evaluates to true, and because of the short-circuiting property of the logical || operator, the entire expression evaluates to true, and hence, "True" is printed. In the second if condition, rem(n, r) is called first and since r=0, n%r results in a floating-point exception.

Now, in the release (optimized) build, there are many kinds of optimizations that the compiler enables. There is something called "constant folding" wherein the compiler takes expressions whose values can be calculated at compile time and replaces them with the result of the calculation directly.

Another such optimization is "common subexpression elimination". Here, duplicated calculations are rewritten to calculate once and duplicate the result.
So, in this sample code, the values of n and r are available at compile time, so, the expression r==0 is replaced with true, thus evaluating the entire first if condition to true. Then using common subexpression elimination, the first if condition & second if condition are found to be the same by the compiler, probably because the logical OR (||) operator is commutative. So, the result of the first if condition is sent to the second if condition, as a result of which the second if also evaluates to true, and thus, "True" is printed twice.

When Line 1 is commented and Lines 2 & 3 are present

Now, the values of n and r are not known at compile time. So, the compiler is unable to apply any of the above said optimizations. Hence, as usual, each line of the code is executed step-by-step, thus giving "True" from the first if condition and floating-point exception from the second if condition.
So, the output is same in both the debug and release builds.

Achutosh Kumar Singh 19CS30008 __/__/____

The difference in output by changing line 1 to line 2 & 3, even though we provide the same input, is because in the first case, the values were known at compile time, so the compiler could substitute those values in the expressions and apply optimizations.
But, in the second case, the values are not known at compile time, instead, they are provided at runtime, so, no optimization can take place from the side of the compiler.

Guideline:-
While executing the code in a release (optimized) build, the compiler applies some optimizations. For example, due to commutativity of logical OR (||) operator, the expressions if (rem(n,r)||r==0) and if (r==0||rem(n,r)) are treated as the same. But, it is our duty to check that within one of the conditions itself, there is no exceptional behaviour that is happening. Thus, it is always advisable to build the program in a debug build before building using a release build, because in the debug build, we will get to know if there is any bug or exception, which may get skipped in the release build.

Ashutosh Kumar Singh 19CS30008 __/__/____

4.

function name	Behaviour	Justification & Comments.
f1()	Bat Segmentation fault (core dumped) First line of output is correctly printed, after that a runtime exception occurs.	First of all "Bat" is treated as a constant string (rvalue) during the assignment, but the LHS is not const. So, when we try to change str[0] to 'c', we get a runtime error.
f2()	Compilation error	Here, since the RHS is const, we correctly declare the LHS as const too. So, str is a char pointer pointing to a constant string. So, trying to change str[0] gives a compilation error as *str is constant and read-only.
f3()	Compilation error	When we write: char * const str = "Bat", here str is a constant pointer, pointing to the string "Bat", so we cannot change this constant pointer to point to anything else. So, trying to make str point to "Rat" using str = "Rat" results in a compilation error.

Ashutosh Kumar Singh 19CS30008 __/__/____

Function name	Behaviour	Justification & Comments
f4()	Bat Cat Rat Here we get the correct & expected output	The function strdup creates a copy of the string passed to it & returns a pointer to the same. So, here, neither the string, nor the char pointer pointing to the string is constant, so we can change str[0] & we can also change the string that str points to.
f5()	Compilation error	Here, using const char *str = strdup("Bat"), the string pointed to by str is made constant, so we cannot change the contents of the string. Hence trying to do str[0] = 'C' gives a compilation error.
f6()	Compilation error	Here, the first line is :- char * const str = strdup("Bat"); So, here the pointer is made constant, i.e., we cannot make the pointer str point to some other string. So, when we try to make str point to a string "Rat" using the line :- str = strdup("Rat"); we get a compilation error.

Ashutosh Kumar Singh 19CS30008 __/__/____

Guidelines :-

- Any constant value on the right side (here strings) should also be stored in a const variable on the left side, so as to maintain uniformity.
- One should understand when which item is constant. For example const char *str makes the string pointed to be str as constant, whereas, char *const str makes the pointer str itself as constant. Hence, one should not change these items which are constant.
- Character strings are by default treated as constants. That is why one can use the strdup method to make a copy of a string. The strdup method should be used while assigning strings as char pointers, as here, we can change both the contents of the string, as well as the pointer itself.

(Continued)

→ As a result of this, we will come to know about errors during compilation itself, & we will not have to face any runtime error.

Ashutosh Kumar Singh 19CS30008 __/__/____

5.

Line No.	Behaviour	Justification & Comments
Line 01	Compilation error	The LHS is an non-const lvalue reference of type 'int&', whereas RHS is an rvalue of type 'int'. This is why we get a compilation error. Simply 'int&' cannot be bound to a constant or temporary value (as in this case), because these cannot be modified. (here the function e returns a constant temporary literal.)
Line 02	Compilation error	Again, we are trying to bind a non-const lvalue reference of type 'int&' to an rvalue of type 'int', which gives a compilation error. The function f too returns an integer literal.
Line 03	We get an output as someone can predict. Values of a and x are same (=10), but the addresses printed are different.	In the function g, a copy of the variable a is passed. Hence, the values of a and x are same, but the addresses are different. Also, since g returns a reference, there is no compilation error this time like the previous cases. (Though the reference returned is a local reference, which is quite dangerous).

Ashutosh Kumar Singh 19CS30008 __/__/____

Line No.	Behaviour	Justification & Comments
Line 04	We get a correct expected output. The values of a and x are same (=10) and their addresses are also the same.	In the function h, the parameter x is a reference, so the value & address of x are same as that of a.
Line 05	Compilation error occurs at line 01, so this line has no meaning.	Line 05 is dependent on line 01, and since line 01 gives a compilation error, control will not reach to line 05 & it makes no sense to talk about line 05.
Line 06	Due to compilation error at line 02, line 06 has no meaning.	Similar to the above case, here line 06 is dependent on line 02, so control won't reach here & we won't get any output.
Line 07	Segmentation fault (core dumped) which means a runtime error.	Here we try to access rr, which is the reference of a local variable (returned from function g). So, as soon as the variable went out of scope, this memory is no longer variable, hence, accessing it causes a segmentation fault.

Ashutosh Kumar Singh 19CS30008 __/__/____

Line No.	Behaviour	Justification & Comments
Line 08	Correct expected output. The values of a, x, rrr are same (=10), and their addresses are also same.	In the function h, we had the reference as a parameter & hence returned that reference. So, basically, x and rrr both are references to a, thus having the same value & address as a.
Line 09	The values of a and x printed are same, but the addresses 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. As a is passed by value, x is a copy of a, so, the addresses are different.
Line 10	The printed values of a and x are same, and their addresses are also same.	The issue of a constant local value is solved by using a constant reference. Since the parameter of the function f is a reference, the address of x is same as a.
Line 11	The values of a and x are same, but their addresses are different.	Since it is pass by value, the addresses of a and x are different, and the values are same (=10), but as we return the reference to a local variable, it is not safe.

Ashutosh Kumar Singh 19CS30008 __/__/____

Line No.	Behaviour	Justification & Comments
Line 12	The values of a and x are same (=10) and so are their addresses. We get a correct expected output.	The parameter to the function h is a reference to a, so we get same addresses for a and x. Also, this is safe, unlike line 11.
Line 13	Values of a, x, rrvc are same (=10), but all their addresses are different.	x is a copy of a, so addresses of x and a are different. Also, rrvc is a new reference to a constant returned value, so that too has a different address.
Line 14	Values of a and x and rrvc are same. Addresses of a and x are same, but address of rrvc is different.	x is actually a reference to a, so their addresses are same. But the return type of the function f is 'int', so the address of rrvc is different as it holds a constant temporary returned value.
Line 15	Segmentation fault (core dumped) - runtime error	Here, since the function g returns the reference of a local variable, similar to what happens in line 07, here too we get a runtime error on accessing it.

Ashutosh Kumar Singh 19CS30008 __/__/____

Line No.	Behaviour	Justification & Comments
Line 16	Correct expected output - The values of a, x, rrrc are same (=10), and, their addresses are also same.	The function h takes a reference as parameter & returns the same reference, so the addresses of a, x and rrrc are the same.
Line 17	Compilation error.	e(a) is an rvalue. It is a constant value returned by a function. But, there should be an lvalue on the left side of the assignment operator. So, we get a compilation error.
Line 18	If we consider line 18 to be dependent on 17, then because of compilation error in line 17, we won't reach to line 18. But treating line 18 independently, we get the value & address of a as output.	If line 18 is considered along with line 17, then we get a compilation error in line 17 itself. But, if taken independently, then it is quite trivial.

Ashutosh Kumar Singh 19CS30008 __/__/____

Line No.	Behaviour	Justification & Comments
Line 19	Compilation error	similar to what happens in line 17, here too we have an rvalue, but we need an lvalue on the left side of the = operator.
Line 20	With line 19 - no meaning because of compilation error. Without line 19 - value and address of a are displayed.	Explanation is exactly same as in case of line 18.
Line 21	g is called. Value & address of x are displayed (diff. from a), followed by a segmentation fault (runtime error).	The parts till where g is called & x is printed is trivial. Now, we try to assign the value 3 to the reference to a local variable, which causes a segmentation fault.
Line 22	Due to segmentation fault in line 21, control does not reach line 22.	Quite trivial. (similar to 18 & 20).

Ashutosh Kumar Singh 19CS30008 __/__/____

Line No.	Behaviour	Justification & Comments
Line 23	The function h prints value & address of x (same as a) & then the value of 4 is assigned to a.	Since the returned value from function h is also a reference to a, assigning 4 to it, sets the value of a to 4.
Line 24	Value of a is printed as 4, & address is same as before.	The value of a changes due to :- h(a) = 4, but the address obviously remains the same.

Guidelines

- Never return a reference to a local variable, as that memory is unallocated after we return from that function.
- While assigning a constant or tempory value to a reference, keep the lvalue a const reference.
- Maintain the fact that there should be an 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.