Nakul Aggarwal
19CS10044

DATE: __/__/__
PAGE: _____

Software Engineering
Assignment 01

① The bug is in the 'if' condition written in the main function.

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

The problem with comparing two floating point numbers is that it does not produce correct output due to internal precision errors in rounding up these numbers.

A small error in rounding off can be enough for the condition "a == sqr(b)" to be wrong. (Eg- $2 \neq 2 + 10^{-50}$). The best fix is to refrain from using '==' operator and instead calculate the absolute difference b/w the two numbers, and compare it with a very small number, say 10^{-6}. If the difference is less than the threshold, the statement must be true.

Nakul Aggarwal
19CS10044

DATE : __/__/__
PAGE: _____

if (a == sqr(b))
↳ if (fabs(a - sqr(b)) < 10e-6)

An important guidline to avoid such bugs is to never use equality operator to compare two floating point numbers. Infact, similar bugs may also arise if '>' or '<' operators are used. So make it a rule to first check equality of two floating-point numbers by the above method and then if they turn out to be unequal '>' or '<' signs can be used. Or instead "a - b > 10e-6" and "b - a > 10e-6" can be used in place of "a > b" and "b > a" for floating-point nos. 'a' and 'b'.

Nakul Aggarwal
19CS10044

DATE: __/__/__
PAGE: _____

(2) The program would not output anything to the console and give a segmentation fault.

No this is not what the developer intended. He must have intended either of the following.
(a) He might have wanted to check if p was a NULL pointer. He might have typed "p == 0" as "p = 0".
(b) He might have wanted to check if the value stored in p was 0. He might have typed "*p == 0" as "*p = 0".

The first intention is more likely because dereferencing a NULL ptr. is lethal. The condition statement "p = 0" is syntactically correct. It assigns p a NULL value and the condition becomes false and hence "No value" is not printed. Then it goes to the else statement. It is here

Nakul Aggarwal
19CS10044

DATE: __/__/__
PAGE: _____

that causes a segmentation fault. 'p' is now a NULL ptr. and dereferencing p means trying to read an illegal memory location. So the bug is the condition statement that "assigns" instead "checking equality".

Correction:

```
if ( p == 0)
    cout << " No Value" << endl;
else
    cout << *p << endl;
```

The following guidlines can be followed to avoid such bugs and improve the quality.

(a) Always check if a pointer is NULL before dereferencing it. Handle the NULL-ptr exception separately.

Nakul Aggarwal
19CS10044

DATE: __/__/__
PAGE: _____

(6) A common mistake is to write '==' operator as '='. In special cases, like when equality is to be checked with zero, avoid using these operators and use logical operators instead.

```
if(p == 0)   ≡   if(!p)
if(p != 0)   ≡   if(p)
```

This makes conditional statements crisp and less error prone.

Nakul Aggarwal
19CS10044

DATE: / /
PAGE:

(3) • Case I (line 1 uncommented)

Consider the first conditional statement.

```
if ( x == 0 || sum(x,x))
```

This statement evaluates to true in both the builds. The evaluation of this condition has nothing to do with the type of build. It is because of "short circuiting evaluation". For the logical operator '||' to be true, it is enough if even one of the two sub-conditions are true. If the one on the left of '||' is true, it need not check the other one. Here, "x == 0" is true and hence the condition "x == 0 || sum(x,x)" evaluates to true without even knowing what exists to the right of "||". Since no call is made to the "sum" fn here, no error occurs.

Nakul Aggarwal
19CS10044

DATE: __/__/__
PAGE: _____

Now consider the second conditional statement.

```
if (rem(n,n) || n==0)
```

The result in the debug build is straight-forward. Due to short circuiting evaluation, it first attempts to compute rem(n,n). Inside the f^n "rem", n%n operation is invalid because n is equal to 0. Hence the floating point exception occurs.

Now when the code is built in the release mode, the compiler performs many optimizations on the code; including replacing variables with hard coded values, removing redundant f^ns that are not called etc (obviously all these are done when it is possible / correct to do so). Eg - in the code fragment -

```
int n = 100;
cout << "value: " << n;
```

the "cout" statement might be

Nakul Aggarwal
19CS10044

DATE: / /
PAGE:

optimized to -

cout << "value : " << 100;

If the code is built in release mode, the optimizer might eliminate the variable 'n' from the program completely & replace all places in the source code where variable 'n' is mentioned with its value 100. [But of course this is possible only if 'n' is initialised in the source code.] — (*)

So in the condition statement "sum(n, n) || n == 0", the optimizer optimizes it simply to "sum(15, 0) || 0 == 0". Now the compiler has already found the true sub-condition "0 == 0" and hence it optimizes again by avoided call to the "sum" f^n. Hence no error occurs and "True" is printed.

Nakul Aggarwal
19CS10044

DATE: __/__/__
PAGE: _____

- Case II (line 1 commented)

The result in the debug build will naturally remain the same. The same reasoning applies as in case I. In release build, however, there is a change. Refer to the starred statement on the last page inside square-brackets. In this case, the variables 'n' and 'x' where declared but not initialized. There values were taken from the user during run-time. Since the compiler does not have any values for 'n' and 'x' it does not do any optimization like it did in the last case. Therefore it remains pretty much same as the debug build in this case & yields the same result as the debug mode gave. Now when the user inputs 0 as the value of 'x', due to short circuiting evaluation in the second

Nakul Aggarwal
19CS10044

DATE: __/__/__
PAGE: _____

condition statement, the function "rem" is called (in both the builds) and error occurs.

- Guidlines to avoid such bugs & improve the quality of code-

(a) In functions like "rem" that can throw exception for certain inputs, handle the special cases separately using control statements. Like add a line " if (x==0) return -1; " before any calculation is attempted. Eg- in a function 'tan(x)' that returns tan(x), the special case of $x = \pi/2$ must be handled before 'tan(x)' fn in <cmath> file is called.

(b) Another practice can be to handle the exceptions "gracefully" by using try catch, throw statements/keywords. They improve the style of the code and make it more readable "exit()" can also be used.

Nakul Aggarwal
19CS10044

DATE: __/__/__
PAGE: _____

(4)

Function Name	Behaviour	Justification & Comments
f1()	compilation error	char* str = "Bat"; str = "Rat"; Conversion of string literals to char* is illegal. The first statement can be corrected by using const keyword. The second statement is wrong due to the same reason. The above conversion is forbidden to prevent any possible change in the characters of the const-string.
f2()	compilation error	str[0] = 'C'; It attempts to assign a new value to the read-only location. The statements that

Nakul Aggarwal
19CS10044

DATE : __/__/__
PAGE: _____

		were wrong in the last function are corrected by changing 'char*' to 'const char*'.
f3()	compilation error	char* const str = "Bat"; str = "Rat"; The above two statements are wrong due to the same reason as in f1. 'const' keyword after the asterisk(*) indicated that the variable 'str' is constant. It does not ensure the read-only characteristic of string literals. So essentially here too a string constant is being converted to 'char*' which is forbidden.
f4()	CORRECT	The "strdup" function creates a copy of the string literal passed

Nakul Aggarwal
19CS10044

DATE: __/__/__
PAGE: _____

		as argument and then returns the ptr. to the first char of the string. The characters in the 'str' string can be read & overwritten because its type is "char*", not "const char*". Ly, the value of 'str' can be changed to another ptr (as in str = strdup("Rat")) because its type is not "char * const str".
f5()	compilation error	str[0] = 'c'; 'str' is a const char ptr. This statements tries to assign new value at a read-only location which is forbidden.
f6()	compilation error	str = strdup("Rat"); similar bug as in f5 but here not the string but 'str' ptr. is const. ∴ It cannot be assigned to a new value.

Nakul Aggarwal
19CS10044

DATE: __/__/__
PAGE: _____

The following guidlines might help in maintaining a good quality of code.

(a) Instead of C-strings, string objects of <string> header file can be used for a far more intuitive initialisation / instantiation of strings. String objects make the handling of strings much more simpler that reduces the chances of errors. String objects can be handled as easily as other primitive data types like int, double, char.

(a.1) Given we have 'char pet[40];"
pet = "Dog" is intuitive but invalid. Instead we will have to do "strcpy(pet, "Dog")".
string objects have the ability to assign data with intuitive notation.

```
string str;
str = "Hello";
str = "Bye";
string str = "KGP";
```

Nakul Aggarwal
19CS10044

DATE: __/__/__
PAGE: _____

All four instructions are valid in the same block of code. C-strings prohibit re-assignment of strings :-

* char * str = "Bat"; // invalid.
* char str[] = "Bat"; // valid
  str = "Rat"; // invalid.

(a.2) The need to use f^n's like "strdup", "strcmp", "strcpy" in C-strings not only harms the readability of the code but also creates chances of mistakes because the signatures of these f^n's can't be crammed. In this case, using overloading operators ('=', '+', '+=', '==', '!=', '>', '<' etc) is far easier.

Nakul Aggarwal
19CS10044

DATE: __/__/__
PAGE: _____

(5)

Line No.	Behaviour	Justificat^{n} & Comments
Line 1	Compilat^{n} error	The initialisat^{n} aims to initialize non-const reference of type 'int&' from an r-value. The function call to "e" is alright but it returns a local 'int' value stored in a temporary address. Since the return variable is temp-orary, reference must be of 'const' type.
Line 2	Compilat^{n} error	Here too the initialisat^{n} aims to initialize a non const ref. of type 'int&' from r-value. The return type of "f" is 'int' which returns a temporary variable. Reference to the return-ed value must be made 'const' as in the first line.

Nakul Aggarwal
19CS10044

DATE: __/__/__
PAGE: _____

Line 3	compilat^{n} error	The bug is in the function definit^{n} of "g". It tries to return reference to a local variable. Once a f^{n} returns a value or terminates, its stack-frame is deallocated. ∴ returning ref. to a local variable which no longer exists is fatal. The f^{n} signature can be changed to "int & g (int & x)" for correct^{n}.
Line 4	CORRECT	The f^{n} "h" is called by passing variable 'a' by ref. Inside the f^{n}, all instruct^{n}s are perfectly valid including the return statement. This is because the 'x' in the f^{n} is now not a variable local to the f^{n} "h" but is instead local to the 'main' f^{n}. This variable can be returned by ref. without

Nakul Aggarwal
19CS10044

DATE: __/__/__
PAGE: _____

		giving the error that the third line gave.
Line 5	CORRECT	All the four instructions
Line 6		are syntactically correct.
Line 7		All the variables mentioned
Line 8		in the print statements ('xvv', 'xxv', 'xvx', 'xxx') are declared in the scope.
Line 9	CORRECT	Line 9 and 10 implement the
Line 10		correct fn's that were proposed for line 1 and 2 respectively. The return types of fn "e" and "f" are 'int', that return a temporary variable that must be initialized to a 'const' reference of type 'int &'.
Line 11	compilation error	The bug is due to the same reason as in line 3. The fn "g" tries to return reference to a local variable of "g" which is fatal. The fn signature should be "int & g (int & x)".

Nakul Aggarwal
19CS10044

DATE: / /
PAGE:

Line 12	CORRECT	The justification is the same as the one given for line 4. Line 12 is different only because of 'const' keyword. 'rrrc' variable is a const-reference to variable 'a'. This means it can ~~also~~ only read the data and address of 'a' but not over-write/reassign it.
Line 13	CORRECT	All four instructions are
Line 14		syntactically correct. All the
Line 15		variable mentioned in the
Line 16		'print' statements ('rvvc', 'rrvc', 'rvrc', 'rrrc') are declared in the scope.
Line 17	compilation	These statements aim to put
Line 19	error	rvalue as the left operand of assignment operator '=', where an lvalue is reqd. The fn's "e" & "f" return an 'int' value stored in temporary address. These temporary variables cannot be assigned new values.

Nakul Aggarwal
19CS10044

DATE: __/__/__
PAGE: _____

Line 21	compilat^{n} error	All instructions that call the f^{n} "g" will produce an error. This f^{n} tries to return ref. to a local variable which is illegal. The f^{n} signature should be "int& g(int& x)", i.e., similar to that of "h". This will also make line 21 correct unlike line 17, 19; by the same justificat^{n} given for line 23 below.
Line 23	CORRECT	First of all, the signature and definit^{n} of f^{n} "h" is correct as per the justificat^{n}s given for lines 4, 12. It returns the variable local to main f^{n} by reference. In this case, 'h(a)' is nothing but an alias/synonym of 'a' returned by "h". ∴ It is valid to assign a value to it. This statement changes value of 'a' from 10 to 4.

Nakul Aggarwal
19CS10044

DATE: __/__/__
PAGE: _____

Line 18	CORRECT	All four statements aim to print the value & adress of 'a' after some attempted transformations. 'a' is a variable local to 'main'. All statements are syntactically correct because the variable 'a' is in scope.
Line 20		
Line 22		
Line 24		

The following guidlines may come in very handy to write a good quality code in such situations.

(a) Refrain from passing parameters of built-in type (int, char, float, double etc) by reference; unless some very specific functionality has to be realised, like swapping two values

(b) Never "return by reference" unless the returned variable was passed to the fⁿ by reference. If some operations are to be performed on UDTs by that fⁿ, a dummy variable local to the caller fⁿ can be passed to the caller by reference

Nakul Aggarwal
19CS10044

DATE: __/__/__
PAGE: _____

and all the changes on the "actual parameter" of the callee will be reflected on the variable local to the caller. In this case no return statement is needed at all, let alone return-by-reference.

(c) Make it a practice to enclose doubtful segments of code in try-catch blocks. This makes debugging fairly easy.