Assignment - 1

CS20006 / CS20202

Name: Rohit Ranjan
Roll : 20CS30066

Q1.

No. The following program is not correct since it does not calculate the correct totalarea. The bug lies in the fact that the macro of pi is assumed to be an integer which leads to loss of precision.

— — —

Fix: #define pi (22.0/7)

(instead of #define pi (22/7))

— — —

~~No~~

Yes. The same bug can repeat if area is defined as an inline function. This is because the macro of pi is still assumed as int and radii is also an int so loss of precision still takes place.

Q2.

The bug is that constant integer a in the inc() function should not be incremented. The fixed code:

— — —

```
#include <iostream>
using namespace std;
const int inc (int a) { return ++a; }
```

Rohit Ranjan
20CS30066

```
int main() {
    cout << inc(5) << endl;
}
```

— — —

(return a++) has been fixed to (return ++a) based on the assumption that inc should return a successor of a.

Returning a const value is a good idea especially when we ~~not~~ return references which can be used as an lvalue. Returning a const reference will protect the pointee to be unchanged even if it is accidentally passed as an lvalue.

Q3.

Case 1: Line 1 is uncommented

Unoptimised: In this case, gcc cannot make any structural changes and code is compiled in the same flow as given. Hence in the first or condition, (n == 0) is already true and second ~~or~~ expression to ~~or~~ is not checked. However, in the second if condition, when the expressions are checked from left to right, rem(15,0) causes an operation 15 % 0. Modulo is basically division with remainder. This modulo operation with 0 causes the Floating Point Exception.

Rohit Ranjan
20CS30066

Optimised: At Here, the compiler recognises that r is a constant integer initialised to 0 and it cannot be changed. In the light of this new information, the if conditions can be always set to true during optimisation, because (r == 0) is always true and consequently the or condition is always true. This causes the removal of all calls to the rem function in compiled code, hence hiding a possible modulo with 0 ~~error~~ error.

Case 2: Line 1 is commented

Unoptimised: This is the exact same as the unoptimised error in case 1. The input of 0 into r, causes a modulo operation with 0 when rem(n, r) is called leading to the exception.

Optimised: Here, the variable r is just an integer and not constant. It can take any value during runtime according to input. Hence, compiler cannot optimise the if conditions. Again, in the second if condition, left to right computation first calls rem(n, r) leading to Floating Point Exception.

Rohit Sanjay
20CS30065

Guideline: Programs should be ideally compiled with -O0, i.e. no optimisations and -Wall and -g flags during development phase to avoid hiding any errors in the code.

Q4.

Yes. There is a bug in the following program. For function and overload resolution, function parameters are read from left to right. Hence, default parameters must be towards the right. There must not be some non-default parameter to the right of any default parameter. This bug happens here in given code:

```
int f1(int a=10, float);
```

This causes f1(10.4) to print "10 30" as 10.4 is converted to int and allotted to variable a according to the prototype:

```
int f1(int, float b=30);
```

Q5. The function is as follows:-

```
int& min(int& c, int& d)
{
    if (c<d) return c;
    else return d;      // WLOG d is min if (c==d)
}
```

Q6.

a.) Both of the functions are legal here.

b.)

Q6. a.) No, this set is not legal because compiler sees (int ptr[]) and (int * ptr) as the same. This leads to both function definitions having same name and signature. Such repeated definition is not legal.

b.) No, this set is not legal only because of the second definition. Whenever declaring a multi-dimensional array as formal parameter, it must have bounds for all dimensions except the first.

c.) No, this set is not legal because both overloads have the same name and signature. This is seen as redefinition of function and not overloading; which is in turn illegal.

~~d.) No, this set is~~

d.) This set of function definitions is legal. However, if not treated carefully, it will lead to ambiguity of overloading for calls of the form fun(x, y) where x, y are integer variables

e.) This set is legal because a pointer stores a memory address while a reference can alias. This

Rohit Ranjan
20CS30066

causes of both functions to have same name but different signatures. This is legal overloading.

Q7.

a.) This set is invalid since (Int operator+(Int a, Int b)) must have an argument of all class or enumerated type and here the typedef just defines an alias to a system-defined type. We must wrap int into a class here to make the set valid.

b.) This set is valid because system-defined type int has been wrapped inside a user-defined struct MyInt and all arguments to the operator are objects of this type.

c.) This set is valid because even if return type is a system-defined type, the arguments are of user defined class.

Q8.

The output of the given code as is is: 5

The bug in functionality is due to a pass by value and not by reference. What happens because of this is that the increment occurs to a local copy of MyInt a and it is consequently not reflected in the variable a in main function.

There can be two ways to achieve desired output.

I.
```
# include <iostream>
using namespace std;
struct MyInt { int v; };
MyInt operator++ (MyInt a, int ) { a.v++; return a;}
```

```
int main() {
    MyInt a;
    a.v = 5;
    a = a++;          // assigning returned value to a itself.
    cout << a.v;
}
```

II.
```
# include <iostream>
using namespace std;
struct MyInt { int v; };
void operator++ (MyInt &a, int) { a.v++; }
                  //changes to operator func.

int main() {
    MyInt a;
    a.v = 5;
    a++;
    cout << a.v;
}
```

Both these modifications will give correct output of 6.

Rohit Ranjan
20CS30066