Software Engineering

Assignment 1

Vishal Subramanyam

20CS10081

CS20006

Problem 1

Problem 1

```cpp
1  #include <iostream>
2  using namespace std;
3  #define pi (22/7)
4  #define area(r) (pi * r * r)
5  int main()
6  {
7      int radii[] = {3,4,5};
8      double totalarea=0;
9      for(int i=0; i<3; i++) totalarea += area(radii[i]);
10     cout << totalarea << endl;
11     return 0;
12 }
```

Listing 1.1: Original program

On line 3, **pi** is a manifest constant. It is processed by the preprocessor that performs textual replacement of these constants and macros. For example, if we take the following piece of code:

```cpp
#define pi (22/7)
...
cout << pi << endl;
```

When the GNU *cpp* program (C Preprocessor) is run on the above code, it produces the following output:

```cpp
cout << (22/7) << endl;
```

As can be seen, the preprocessor simply replaces whatever text is next to the name of the constant directly (even with the parentheses). Now, in our program, we have two **#define**s that are of interest. The second one is a **macro** that performs the same textual replacement, but now, it takes a parameter. This parameter is replaced inside the body of the macro whenever the macro is placed in the file along with the parameter passed to it. For example,

```cpp
#define area(r) (3.14 * r * r)
...
cout << area(2) << endl;
```

The preprocessor output for the above is given below:

```cpp
cout << (3.14 * 2 * 2) << endl;
```

So the **area(2)** macro was replaced by the text next to its name in the **#define** along with parameter substitution. Using this knowledge, we can see that the preprocessor output of the original program would be:

```cpp
...
using namespace std;
int main()
{
    int radii[] = {3,4,5};
    double totalarea=0;
    for(int i=0; i<3; i++) totalarea += ((22/7) * radii[i] * radii[i]);
    cout << totalarea << endl;
    return 0;
}
```

The **area(r)** macro and **pi** constant have both been textually substituted.

Output:

```
150
```

THERE IS A BUG IN THIS PROGRAM. What is the bug? When the code tried to divide 22 by 7, the programmer expected to get a rational value of approximately **3.14**. But, in C++, integral values like **22** amd **7** take on the type `int` whenever they are placed in code. The `/` binary operator performs division based on the type of its operands. Since both **22** and **7** are `int`s, this means integer division is performed. This division does **NOT** produce values with a decimal point like **3.14**. Instead, **(22/7)** takes on the value **3**. This is because integer by integer division is expected to produce an integer as output. Therefore, the areas computed will be less than their true values since **pi** has been rounded down to **3**.

How can this bug be fixed? We want to perform floating-point division. So we replace **22** with **22.0**. This ensures this value is treated as a `double` by the compiler, hence triggering double division with the denominator being promoted to the `double` type. The resultant value is of type `double`, hence giving a more accurate value for `totalarea`.

Can inlining area(r) affect this bug? This bug has nothing to do with the way `area(r)` is defined. The bug is because of the `int` by `int` division that is performed resulting in a rounded down value. Regardless of textual substitution (for `\#define`) or function inlining, the division performed will still be integer by integer unless we cast or convert the operands to a floating point type. We can confirm this by replacing the macro with the inline function.

```cpp
#include <iostream>
using namespace std;
#define pi (22/7)
inline double area(int r) { return pi * r * r; }
int main()
{
    int radii[] = {3,4,5};
    double totalarea=0;
    for(int i=0; i<3; i++) totalarea += area(radii[i]);
    cout << totalarea << endl;
    return 0;
}
```

Output:

```
150
```

We can see the output is the same as the previous one. Hence, the bug still exists. We can again resolve this using the same solution as described.

Program with bug fixed:

```cpp
#include <iostream>
using namespace std;
#define pi (22.0/7)
#define area(r) (pi * r * r)
int main()
{
    int radii[] = {3,4,5};
    double totalarea=0;
    for(int i=0; i<3; i++) totalarea += area(radii[i]);
    cout << totalarea << endl;
    return 0;
}
```

Output:

```
157.143
```

Problem 2 Problem 2

```
#include <iostream>
using namespace std;
const int inc(const int a) { return a++; }
int main() {
    cout << inc(5) <<endl;
}
```

The bug is on line 3, with two different errors. One is an obvious compilation error where the code attempts to increment a variable of type `int` with a `const` qualifier. This cannot be done.
Second, the programmer's intention is assumed to be that of getting the incremented value of the actual parameter passed to the `inc` function. Since we have to take `const` arguments, we cannot remove the `const` qualifier from the parameter. Instead, we will change `a++` to a+1. Hence the final program is,

```
#include <iostream>
using namespace std;
const int inc(const int a) { return a+1; }
int main() {
    cout << inc(5) <<endl;
}
```

Output:

```
6
```

Are there any uses to using a return value with `const` qualifier?
For non-class types such as `int`, it does not make any difference at all since the manner in which the return value is used results exclusively in **prvalue**s. For this value category, according to the C++ standard, a `const` qualifier does **NOT** make a difference in the way it's used.
For classes, this can matter. One **use case** is where we want our return object to be immutable. For instance,

```
#include <iostream>
using namespace std;
class Box
{
    int h;
    int w;
    int b;

public:
    Box(int h, int w, int b) : h(h), b(b), w(w) {}

    int GetH() const
    {
        return h;
    }
    int GetW() const
    {
        return w;
    }
```

```cpp
    int GetB() const
    {
        return b;
    }
    void modify()
    {
        h = w + b;
        w = b + h;
        b = h + w;
    }
};
const Box myFunc()
{
    return Box(1, 2, 3);
}

int main(){
    int a = myFunc().GetW(); // ALLOWED
    myFunc().modify(); // ERROR
}
```

This can be useful in cases where we want the returned object to be constant and immutable.

But this practice of returning `const` objects is highly discouraged in modern C++ since **move** semantics are blocked by the usage of a `const` return object. A copy constructor is called instead, since a move constructor would require a non-const reference to the object.

Problem 3 Problem 3

```cpp
#include <iostream>
using namespace std;
int rem(int n, int r) {
    return n % r;
}
int main() {
    const int n = 15, r = 0; //line 1
    // int n , r; // Line 2
    // cin >> n >> r; //Line 3

    if (r == 0 || rem(n, r))
       cout << "True" << endl;
    else
        cout << "False" << endl;
    if (rem(n, r) || r == 0)
        cout << "True" << endl;
    else
        cout << "False" << endl;
return 0;
}
```

Listing 3.1: Original program

Output:

```
True
```

```
Floating point exception (core dumped)
```

Listing 3.2: Output with -O0

```
True
True
```

Listing 3.3: Output with -O1

What is happening? What is the optimization that led to the change in output? Is this a bug? The explanation is as follows:
Without any optimizations, the `g++` compiler generates object code that closely mirrors the source code without trying to shorten, simplify or speed up the code much. This results in an executable that has an explicit call to the function. Since the function involves dividing by zero, an exception is caused, and the output is as expected.

But with `O1` level of optimization, the compiler has two ways of detecting redundant code. One is via **inter procedural analysis of pure/const functions**, and the other is via **inlining** the function call, which results in optimizations such as **dead code elimination** (among others) to eliminate or rearrange the inlined function body. Specifically, with respect to **inlining**, since its value is irrelevant to the result of the logical expression, and also has no visible side-effects. If the function had effects like using *I/O* or modifying global memory, then the compiler would be forced to retain the function body after inlining it since eliminating it would violate the "as-if" rule of the C++ standard that expects optimized compiler output to behave *as if* the optimization hasn't happened. So, not executing the function body (either via elimination or rearrangement) also involves eliminating the erroneous division by zero attempt, hence not causing an exception.

With respect to **inter procedural analysis of pure/const functions**; even if our program had inlining disabled, the same difference between O0 and O1 versions of the program would persist. How so? Didn't disabling inlining prevent the optimization of the function body? What happened is that the compiler used interprocedural analysis, i.e. analysis that involves going beyond the current function's boundary to establish optimizations, to realize that the function `rem` is a **pure/const** function.
Pure function: A pure function is one which do not affect anything outside of it's own scope. This means it may read global variables or variables to which it was passed a pointer but it may not write to such variables. It should not read from volatile variables or external resources (such as files).
Const function: const is a stricter version of pure. It tells the GNU compiler that the function body won't reading any data other that of the variables that are passed to it. Data cannot be read by dereferencing a pointer passed to a const function. The only effect a pure or const function can have on a program is it's return value. Therefore, when interprocedural analysis involving pure/const functions is done on the program, the compiler will realise that the function `rem` does not cause any side-effects. Once this happens, similar to the inlining case, the compiler will decide to optimize the function call and may choose to rearrange code since it can still adhere to the "as-if" rule of the C++ standard since the function's evaluation later may not cause a different effect. We can verify this passing the `-fno-ipa-pure -const` and `-fno-inline` options to the GNU compiler when we compile our program with O1 optimization. The `-fipa-pure-const` optimization basically tells the compiler that it is okay to glean information about the pure/const nature of functions and use that for optimization. The `-finline` optimization permits inlining of functions should the compiler wish to do so.

Prepending *no* to both of these options after *f* implies the opposite and hence suitable for our usage.

```
1  $ g++ main.cpp -O1 -fno-inline -fno-ipa-pure-const
2  $ ./a.out
3  True
4  Floating point exception (core dumped)
5  $ g++ main.cpp -O1 -fno-inline
6  $ ./a.out
7  True
8  True
9  $ g++ main.cpp -O1 -fno-ipa-pure-const
10 $ ./a.out
11 True
12 True
```

We can also see that disabling just one of these two optimizations is not sufficient for the division operation to be not skipped. In the absence of IPA, since the `rem` function is simple enough, it will be inlined, and the compiler will be able to optimize the function body.
In the absence of inlining, with IPA, the compiler will realize that the `rem` function is a **const** function and will be able to perform a similar optimization.

But in the second case, where we replace the constant variables with values from the standard input, the compiler can no longer perform that optimization since the compiler has to allocate space for these variables (earlier, they were inlined directly) and pass them to the function. Since the user input is unknown and could potentially be anything, the compiler cannot analyze the effects of `rem` and thus optimizations are not applied to that function.

We can verify all the above by taking a look at the relevant AMD64 assembly code produced by the compiler:

```
1  $ g++ main.cpp -O0 -S
2  $ cat main.s
3  ...
4  main:
5      ...
6    movl  $0, %esi
7    movl  $15, %edi
8    call  _Z3remii
9    movl  $1, %eax
10   testb %al, %al
11   ...
```

On line 9, we can clearly see the name mangled `rem` being called. This results in the floating point error that we see. We also see that actual variables have been allocated on the stack and direct values of the constants have **NOT** been used to optimize anything.

```
1  $ g++ main.cpp -O1 -S
2  $ cat main.s
3  ...
4  main:
5  .LFB1565:
6    .cfi_startproc
7    subq  $8, %rsp
8    .cfi_def_cfa_offset 16
9    movl  $4, %edx
10   leaq  .LC0(%rip), %rsi
```

```
  leaq  _ZSt4cout(%rip), %rdi
  call
   _ZSt16__ostream_insertIcSt11char_traitsIcEERSt13basic_ostreamIT_T0_ES6_PKS3_l@PLT

  leaq  _ZSt4cout(%rip), %rdi
  call  _ZSt4endlIcSt11char_traitsIcEERSt13basic_ostreamIT_T0_ES6_@PLT
  movl  $4, %edx
  leaq  .LC0(%rip), %rsi
  leaq  _ZSt4cout(%rip), %rdi
  ...
  leaq  _ZSt4cout(%rip), %rdi
  call  _ZSt4endlIcSt11char_traitsIcEERSt13basic_ostreamIT_T0_ES6_@PLT
  movl  $0, %eax
  addq  $8, %rsp
  .cfi_def_cfa_offset 8
  ret
  ...
```

We can see that the main function didn't even bother testing the values for equality and has straight up just moved on to the printing portion of the program. This persists even with the `-fno-inline` option.

```
...
main:
.LFB1541:
  .cfi_startproc
  pushq %rbx
  .cfi_def_cfa_offset 16
  .cfi_offset 3, -16
  leaq  .LC0(%rip), %rsi
  leaq  _ZSt4cout(%rip), %rdi
  call  _ZStlsISt11char_traitsIcEERSt13basic_ostreamIcT_ES5_PKc@PLT
  movq
   _ZSt4endlIcSt11char_traitsIcEERSt13basic_ostreamIT_T0_ES6_@GOTPCREL(%
   rip), %rbx
  movq  %rbx, %rsi
  movq  %rax, %rdi
  call  _ZNSolsEPFRSoS_E@PLT
  leaq  .LC0(%rip), %rsi
  leaq  _ZSt4cout(%rip), %rdi
  call  _ZStlsISt11char_traitsIcEERSt13basic_ostreamIcT_ES5_PKc@PLT
  movq  %rbx, %rsi
  movq  %rax, %rdi
  call  _ZNSolsEPFRSoS_E@PLT
  movl  $0, %eax
  popq  %rbx
  .cfi_def_cfa_offset 8
  ret
  .cfi_endproc
...
```

Only after disabling both IPA pure const and inlining, can we see the function call happening and resulting in an error.

```
$ g++ main.cpp -O1 -fno-inline -fno-ipa-pure-const -S
$ cat main.s
...
main:
```

```
.LFB1541:
  .cfi_startproc
  pushq %rbx
  .cfi_def_cfa_offset 16
  .cfi_offset 3, -16
  leaq  .LC0(%rip), %rsi
  leaq  _ZSt4cout(%rip), %rdi
  call  _ZStlsISt11char_traitsIcEERSt13basic_ostreamIcT_ES5_PKc@PLT
  movq
    _ZSt4endlIcSt11char_traitsIcEERSt13basic_ostreamIT_T0_ES6_@GOTPCREL(%
    rip), %rbx
  movq  %rbx, %rsi
  movq  %rax, %rdi
  call  _ZNSolsEPFRSoS_E@PLT
  movl  $0, %esi
  movl  $15, %edi
  call  _Z3remii
  leaq  .LC0(%rip), %rsi
  leaq  _ZSt4cout(%rip), %rdi
  call  _ZStlsISt11char_traitsIcEERSt13basic_ostreamIcT_ES5_PKc@PLT
  movq  %rbx, %rsi
  movq  %rax, %rdi
  call  _ZNSolsEPFRSoS_E@PLT
  movl  $0, %eax
  popq  %rbx
  .cfi_def_cfa_offset 8
  ret
...
```

We can clearly see that `rem` is called. Now, let's test the version which takes user input.

```
$ g++ main.cpp -O1 -fno-inline -S$
$ cat main.s
...
main:
.LFB1541:
  .cfi_startproc
  pushq %rbx
  .cfi_def_cfa_offset 16
  .cfi_offset 3, -16
  subq  $16, %rsp
  .cfi_def_cfa_offset 32
  movq  %fs:40, %rax
  movq  %rax, 8(%rsp)
  xorl  %eax, %eax
  movq  %rsp, %rsi
  leaq  _ZSt3cin(%rip), %rdi
  call  _ZNSirsERi@PLT
  leaq  4(%rsp), %rsi
  movq  %rax, %rdi
  call  _ZNSirsERi@PLT
  movl  4(%rsp), %esi
  testl %esi, %esi
  je  .L10
  movl  (%rsp), %edi
  call  _Z3remii
  testl %eax, %eax
  je  .L11
```

```
28 .L10:
29   leaq  .LC0(%rip), %rsi
30   leaq  _ZSt4cout(%rip), %rdi
31   call  _ZStlsISt11char_traitsIcEERSt13basic_ostreamIcT_ES5_PKc@PLT
32   movq
       _ZSt4endlIcSt11char_traitsIcEERSt13basic_ostreamIT_T0_ES6_@GOTPCREL(%
       rip), %rsi
33   movq  %rax, %rdi
34   call  _ZNSolsEPFRSoS_E@PLT
35 .L12:
36   movl  4(%rsp), %ebx
37   movl  %ebx, %esi
38   movl  (%rsp), %edi
39   call  _Z3remii
40   testl %ebx, %ebx
41   je  .L13
42   testl %eax, %eax
43   je  .L17
44 ...
```

We can see that the function call occurs before the jump instruction involving the second operand, so the call goes through and results in a floating point exception (lines 39-43). **Guideline**: The optimization itself is not a bug since the compiler provides a program that runs "as-if" the optimizations had not happened assuming these exceptions are not taken into account. It's the programmer's job to ensure that mistakes such as division by zero does not occur by writing down the expected range of values for their functions and ensuring the function gracefully handles output outside this range by using tools such as `try-catch` statements. They may also use **assert** statements to verify their assumptions.

Problem 4 Problem 4

```cpp
#include <iostream>
using namespace std;
int f1(int a,float b) {
    cout <<a << " " <<b <<"\n";
    return 0;
}
int f1(int, float b=30);
int f1(int a=10, float );
int main() {
    f1(10,20);
    f1(10.4);
    return 0;
}
```

Listing 4.1: Original program

What is a bug? A bug is an error, flaw or fault in a computer program or system that causes it to produce an incorrect or unexpected result, or to behave in unintended ways. Therefore, we have to determine the programmer's intention before we can determine if the program is buggy. Making some basic assumptions about the program, we assert the programmer's intention by writing down their expected output and their "faulty" reasoning for that expectation. We assert the behavior **IS** a **bug** by showing how the assumed expected behavior deviates from the actual behavior.

Expected output:

```
10 20
10.4 30
```

Faulty reasoning: The programmer expected that the value **10.4** would be passed as an argument to the function `f1` and it will be printed to the standard output as the second line, along with the second parameter which takes on the default value of **30** (line 7). **Actual output**:

```
10 20
10 30
```

The programmer has failed to take into account the fact that the first parameter of `f1` is of type `int`. Now, taking this reference from cppreference.com :

> A *prvalue* of floating-point type can be converted to a *prvalue* of any integer type. The fractional part is truncated, that is, the fractional part is discarded. If the value cannot fit into the destination type, the behavior is undefined (even when the destination type is unsigned, modulo arithmetic does not apply). If the destination type is `bool`, this is a boolean conversion...

This is a legal conversion and is **NOT** undefined behavior. Some compilers may not warn the programmer about this unless a special flag is passed. For `g++` compiler, the flag `-Wconversion` needs to be given at compile-time if a warning is required regarding this implicit floating-point to integer conversion.

Thus, the **10.4** in `f1(10.4)` is implicitly truncated to **10**, therefore giving us the actual output that we saw earlier.

Problem 5 Problem 5

```
1 int main() {
2     int c,d;
3     cin >> c >>d;
4     cout <<c <<" "<< d<<"\n";
5     min(c,d) = 1;
6     cout <<c <<" "<< d<<"\n";
7     return 0;
8 }
```

Listing 5.1: Given program

Requirement: The minimum of `c` and `d` should be assigned the value 1.

Assumptions: Since the problems statement does not mention what the function should do in the case where `c` and `d` are equal, we'll assume the first argument of the function will be set to **1** in the given program.

Solution:

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

Listing 5.2: definition of `min`

Explanation: Since we need to manipulate the variables directly, we use references to refer to the actual arguments so that the function can return a reference to the arguments, thus enabling the usage of the return value of `min` as an **lvalue** (line 5 of listing 5.1). Thus, when `min(c,d)` is called, we get a reference to the variable with the lower value. Assigning **1** to that changes the value of the variable that the reference is tied to.

Problem 6 Problem 6

When multiple definitions are given for the same function signature, this causes a problem with linking since the linker will not know which function to link with a particular function call since there are multiple viable candidates. Therefore, the functions with the same name should have different signatures (excluding the return type). This is called the one definition rule.

6.a Subproblem

```
int fun(int *ptr, int n) {...};
int fun(int ptr[], int n) {...};
```

In C++, excluding certain cases, in expressions, an array decomposes to a pointer that points to the first element of the array. This is the case in function parameters, where an array parameter is taken to be a pointer. Thus the two function signatures are exactly the same, but each signature is followed by a definition. This violates the one definition rule that we saw earlier and is, hence, **NOT LEGAL**.

6.b Subproblem

```
1 int fun(int **ptr, int n) {...};
2 int fun(int ptr[][], int n) {...};
```

This set of function definitions is **ILLEGAL** because the second function in this set is **NOT** a valid C++ function since a parameter that takes a multidimensional array should have bounds for all dimensions except the first. This is because the compiler needs to know how to lookup a certain index by performing offset calculations.

6.c Subproblem

```
1 int fun(int x, int y);
2 static int fun(int x, int y);
```

Referring to ISO C++ 20 standard (9.11-6),

> If two declarations declare functions with the same name and parameter-type-list (9.3.3.5) to be members of the same namespace or declare objects with the same name to be members of the same namespace and the declarations give the names different language linkages, the program is ill-formed;...

Declaring a function without any linkage specifer is the same as declaring it with `extern` linkage. This is done on line 1. But, on line 2, we specify a different linkage specifier, i.e. `static`. Thus, we are violating the language standard, and thus this set of function declarations is **illegal**.

6.d Subproblem

```
1 void fun(int x, int y) {cout<<"f1\n";}
2 void fun(int &x, int y) {cout<<"f2\n";}
```

It is obvious that the compiler cannot choose between the two functions defined above when it has to link a function call to the function code. This is because non-const integral variables passed to `fun` as the first parameter can be either be passed by value as in line 1, or have a reference attached to them, as in line 2. Thus this set of function definitions is **ILLEGAL**.

6.e Subproblem

```
1 void fun(int *x, int y) {cout<<"f1\n";}
2 void fun(int &x, int y) {cout<<"f2\n";}
```

This set is **LEGAL** since the two function signatures are different and this results in **function overloading** in C++. The linker knows which function to link against since function overloading involves name mangling.

Problem 7 Problem 7

7.a Subproblem

```
typedef int Int;
Int operator+(Int a,Int b) { Int c; c = a + b + 5; return c;}
```

`Int` is just a synonym for `int` and is equivalent to it. Hence, this is **NOT** valid since this involves redefining the **built-in** + operator for the **primitive** `int` datatype. This is not allowed in C++.

7.b Subproblem

This **IS VALID**, since there's no such restriction as above on overloading operators for a `struct`.

```
#include<iostream>
using namespace std;
typedef struct _int { int v; } MyInt;
MyInt operator+(MyInt a,MyInt b) {MyInt c; c.v = a.v+b.v+5; return c;}
int main(){
    MyInt a{5};
    MyInt b{10};
    cout << (a + b).v << endl;
}
```

Output:

```
20
```

7.c Subproblem

This form of overloading is valid since it does not attempt to override any of the built-in operators and is exclusively defined for a specific struct. This is clearly legal as is exemplified in the below program.

```
#include<iostream>
using namespace std;
struct MyInt { int v; };
int operator+(MyInt a,MyInt b) { return a.v+b.v+5; }
int main(){
    MyInt a{5};
    MyInt b{10};
    cout << (a + b) << endl;
}
```

Output:

```
20
```

Problem 8 Problem 8

```
#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++;
    cout<<a.v;
}
```

Output:

```
5
```

We want to increment the value inside `a`. To do that, we have to use a reference to the actual struct, otherwise, the members of the struct will simply be copied to the parameter of the operator overload and all increments perforemd there will happen on a local copy of the struct instead of our original struct. Below is a modified version of the program that does what we want, where we use a C++ reference to directly refer to the struct:

```
#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++;
    cout << a.v;
}
```

Output:

```
6
```