Homework Assignment 1

William Jagels

October 8, 2016

I. Q&A

I.1.

Code Block 1: badcode.c

```c
int main(int argc, char * argv[])
{
    char * type = NULL, * op1 = NULL, * op2 = NULL;
    int a = 0, b = 0;
    if (argc != 3 && argc != 4)
    {
        print_usage();
        exit(0);
    }
    type = argv[1];
    op1 = argv[2];
    op2 = argv[3];
    a = atoi(op1);
    b = atoi(op2);
    switch (type)
    {
        case "1":
            printf("Multiplication result is %d\n", mymul(a, b));
        case "2":
            printf("Division result is %d\n", mydiv(a, b));
        case "3":
            myabs(a);
            printf("Absolute value is %d\n", a);
        case "4":
            printf("Concatenation result is %s\n", myconcat(op1, op2));
    }
    return 0;
}
void print_usage()
{
    printf( "Usage: \"./a.out operation_type operand_1 [operand_2]\"\n"
            "\t operation_type :\n"
            "\t\t 1: multiply operand1 and operand2\n"
            "\t\t 2: divide operand1 by operand2\n"
            "\t\t 3: absolute value of operand1\n"
            "\t\t 4: concatenate operand1 and operand2\n");
}
int mymul(int a, int b)
{
    return a * b;
}
int mydiv(int a, int b)
{
    return a / b;
}
void myabs(int a)
{
```

```
48      a = -a;
49  }
50  char * myconcat(char * a, char * b)
51  {
52      char * str = malloc(strlen(a) + strlen(b));
53      strcpy(str, a);
54      strcat(str, b);
55      return str;
56  }
```

Problems with the above

1. All the functions are implicitly defined, which isn't all that safe. Forward declarations should be used.
2. At line 12, `arv[3]` may point past the end of the array.
3. At line 15, the switch statement has no default, making it hard for the user to figure out that they supplied an invalid type.
4. The switch case also won't work because C only alows integers in a switch case. This can be fixed by using `switch(atoi(type))` and removing the quotes around each case number.
5. The switch case also behave erratically without `break;` at the end of each case. All operations after the selected operation will all execute without it.
6. `myabs` does not have any effect as it doesn't return anything. Also, absolute value of a is not always -a. Replacing `myabs` with `abs` from `<stdlib.h>` is a good alternative. This would require deleting line 22 and replacing `a` with `abs(a)` on line 23.
7. `myconcat` allocates a block for the resultant string, but it is never freed when called from `main`.
8. `myconcat` doesn't allocate enough space at line 52, it should allocate `strlen(a) + strlen(b) + 1` bytes because of the null terminator.

I.2.

This code could be useful if `dest` points to a memory-mapped hardware output device. Or it might also be useful in a rowhammer attack.

I.3.

(a) `int a, *b;`
`a` is a normal integer while `b` is a pointer to an integer

(b) `double (fs[])(int);`
`fs` is an array of functions that take an `int` and return a `double`. Sadly, this is not a valid type, because functions can't be stored in arrays. A `*` before `fs` will make this a valid type.

(c) `double *(*f)(double d);`
`f` is a pointer to a function that takes a `double` and returns a `double *`

(d) `int *const p;`
`p` is a constant pointer to an integer. The pointer address can't change, but the data it points to can be.

(e) `int (*cmps[10])(const void *, const void *);`
`cmps` is an array of 10 pointers to functions that take two pointers of type void that point to constants and return `int`.

I.4.

(a)

```
typedef double (*arg_t[])(int);
typedef void *(*ret_t)();
ret_t f(arg_t);
```

(b)

```
void *(*(*foo)(double (*[])(int)))();
```

Code Block 2: Function Pointers

```
#include <stdio.h>
#include <stdlib.h>

typedef double (*arg_t[])(int);
typedef void *(*ret_t)();

void *(*(*foo)(double (*[])(int)))();
double bar(int k) { return 1.0 * k; }
void *baz() { return malloc(10); }

ret_t fin(arg_t a) {
    printf("%f\n", a[0](6));
    return &baz;
}

int main() {
    foo = &fin;
    double (*asdf[])(int) = {&bar};
    free(foo(asdf)());
}
```

I.5.

(a) The compiler actually won't care because function declarations with empty arguments semantically means that the function can take any number of arguments of any type. This can be explicitly overridden by declaring `f` as `void f(void);`.

(b) This is true for the most part, with the exception of floating point numbers. If x is NaN, then `x == x` will always be false. Wonderfully enough, `x == NAN` doesn't even work if x is NaN. `isnan(x)` is the only safe way of checking if x is NaN since `NAN != NAN`.

(c) In some cases, this is true. The C standard only specifies a minimum size of 32 bits for `long`. In the case of a 32 bit `long` and 32 bit `int` there will be overflow because of the sign bit. On my machine, a `long` is 64 bits wide, and does not cause any overflow. A good way to guarantee safety is to use the `long long` specifier, which will always be at least 64 bits wide.

(d) `malloc` is a fairly expensive call, so calling it a bunch of times isn't the best idea. Also, calling `malloc` repeatedly will likely result in fragmented memory, meaning that memory locality will be broken. In addition, in 4-byte aligned systems, this will result in a 25% utilization of the malloc'd blocks.

(e) `typedef` doesn't need to define any <u>new</u> types. A typedef might simply create an alias to an already defined type, as in this nasty typedef: `typedef char* string`