General Instructions: This exam must be turned in to my mailbox in 384 Fitzpatrick Hall on Monday, December 10, 2018 by 4pm. It may not be turned in late. All questions must be worked on and completed individually. You may use online references as necessary, but the answer that you submit must be wholly your own. You may not collaborate with other students (regardless of whether or not they are enrolled in this class). You may, however, speak to the TAs and ask questions if you need clarification on any question.

It is intended that you search for and read official documentation, as well as compiling the code samples on the school’s computers, in order to explore the questions. The questions are designed to expand your understanding of the C++ language, with special emphasis on topics that are practical and are important to know in industry.

Question 1.
(5 points) The following code is an infinite loop and never exits. Why?

```cpp
1 #include <cstdlib>
2 int main() {
3     std::size_t count = 10;
4     while (count-- >= 0);
5 }
```

Question 2.
(5 points) What is a memory leak? Write a short function called `memleak()` that has a memory leak.
Question 3.
(10 points) The following code is written using pass-by-reference. Rewrite `doubleit()` and `main()` so that a pointer is used in place of the pass-by-reference (this is sometimes referred to as “C-style pass-by-reference”, although, strictly speaking, it is actually a form of pass-by-value), but that the output of the `cout` (line 9) is unchanged.

```cpp
#include <iostream>
using namespace std;
void doubleit(double & d) {
    d *= 2;
}
int main() {
    double num = 18;
    doubleit(num);
    cout << num << endl;
}
```
Question 4.
(15 points) Given the following code snippet and its output, provide a `print_base()` function that will **recursively** print the given number in the requested base (up to base 36, which uses the digits 0-9 and letters ‘A’-‘Z’). The function prototype is given for you.

Code:

```cpp
#include <iostream>
#include <cmath>
using namespace std;

void print_base(unsigned long num, int base);

int main()
{
    for (int i = 2; i <= 36; ++i) {
        unsigned long num = pow(2, i - 2) / 2;
        cout << num << " in base " << i << " : ";
        print_base(num, i);
        cout << endl;
    }
    return 0;
}
```

Sample output:

```
0 in base 2: 0
1 in base 3: 1
2 in base 4: 2
... 
64 in base 9: 71
128 in base 10: 128
256 in base 11: 213
... 
2147483648 in base 34: 1D8XQRQ
4294967296 in base 35: 2BR45QB
8589934592 in base 36: 3Y283Y8
```
**Question 5.**
(10 points) You implemented a pre-increment and a post-increment operator in your `Fraction` library. Consider the following two loops:

```cpp
1  for (Fraction f = 0; f < 10; f++) { cout << f << endl; }
2  for (Fraction f = 0; f < 10; ++f) { cout << f << endl; }
```

Both versions produce the same output, and are essentially identical in function. One version, however, may be considered to be more efficient. Which version (pre-increment or post-increment) may be more efficient, and what is your justification for this answer?

**Question 6.**
(20 points) When copying objects, explain the difference between a *shallow copy* and a *deep copy*. Look up and explain what the *Rule of Three* is (for later versions of C++, it is called the *Rule of Five*). Finally, explain how the Rule of Three and the shallow/deep copy concept are related, and when the rule may apply.
Question 7.
(15 points) The following recursive code was written by a beginner programmer, and does indeed function correctly, but it is needlessly verbose:

```cpp
#include <iostream>
#include <cstring>
using namespace std;

bool isPalindrome(const char * s, size_t length) {
    if (length == 0) {
        return true;
    } else if (length == 1) {
        return true;
    }
    else if (length >= 2) {
        if (*s == s[length - 1]) {
            return isPalindrome(s + 1, length - 2);
        } else {
            return false;
        }
    } else {
        return false;
    }
    return true;
}

int main() {
    char buffer[100];
    while (cin.getline(buffer, 100)) {
        cout << (isPalindrome(buffer, strlen(buffer)) ? "P" : "X") << "\t" << buffer << endl;
    }
}
```

Rewrite the `isPalindrome()` function so that the function only contains a `return` statement (similar to the naive `fib()` code of Question 8). You may receive partial credit for partial simplification of the function. *(Hint: Think about how logical operators can be combined.)*
Question 8.
(15 points) We demonstrated in the class lecture how that the $N$th Fibonacci number could be calculated recursively, however at a significant performance cost. The reason for this performance cost is that, for every non-base case function call, two recursive calls are made.

```c
1 unsigned int fib(int n) {
2     return n <= 1 ? n : (fib(n-1) + fib(n-2));
3 }
```

Later, we learned about static variables, and how that they can be used for memoization in functions (including recursive functions). By adding just one static unsigned int to the body of fib(), the function can be completely re-written so as to only require one recursive call instead of two recursive calls, making the function practical for use.

Rewrite the body of fib() using one static unsigned int and one recursive function call. You may create other, helper variables in the body of the function as needed. Partial credit will be given if you can find a solution with two static variables, but full credit will only be awarded if the single-static variable solution is found.

**Hints:** Most online solutions to this problem use a static array of numbers, however that is not the correct solution for this problem. You should only use a single static unsigned int. Also, for context, your solution will not be as short as the ternary solution of the naive, double-recursive solution. Finally, if you are having trouble with this problem, then try writing the two-static variable version first, as it may help you discover the one-static variable solution.