MTH 4300
Spring 2025
E. Fink

Homework 3

---

1. Try to do this problem without actually running the code!

   a. In the code below, which printed memory addresses will be the same as each other? Use the letters prior to the print statements to identify which addresses are the same.

   b. When the six variables are printed out, what will their values be?

```
int f(int & p1, int p2) {
   int local = p1 * 4;
   p1 = 12345;
   cout << "A: " << &p1 << endl;
   cout << "B: " << &p2 << endl;
   cout << "C: " << &local << endl;

   cout << p1 << ' ' << p2 << ' ' << local << endl;
   return local;
}

int main() {
   int x = 8;
   int &y = x;
   int z = y;

   z = f(x, z) + 3;

   cout << "D: " << &x << endl;
   cout << "E: " << &y << endl;
   cout << "F: " << &z << endl;

   cout << x << ' ' << y << ' ' << z << endl;
   return 0;
}
```

2. Try to do this problem without actually running the code!

   Consider the following code, which prints out 5 lines:

```
int main() {
   double x[4];
   cout << &(x[0]) << endl;
   cout << &(x[1]) << endl;
   cout << &(x[2]) << endl;
   cout << &(x[3]) << endl;
   cout << x << endl;
}
```

   If the first line prints `0x34f9dff8b0`, what would the next four lines print (probably)?

3. The following code has some weird behaviors, and at least one thing that looks like a typo but was put in deliberately. What prints out? Explain.

```
int main() {
    int x = 20, y = 30, z = 40;
    cout << (y = ++x) << endl;
    cout << (z = y += 2) << endl;
    if (z = 50) {
        cout << x << " " << y++ << " " << z << endl;
    }
}
```

4. Consider the function $f(n) = \begin{cases} \frac{n}{2}, & n \text{ is even} \\ 3n+1, & n \text{ is odd} \end{cases}$, where both input and output should be integers. For any integer n, you can create a sequence starting with n where each subsequent term is obtained by applying f to the previous term. The *Collatz* conjecture states that, no matter what value of n you start with, the sequence will always eventually include 1.

   For example, starting with $n = 6$: $f(6) = 3$; $f(3) = 10$; $f(10) = 5$; $f(5) = 16$; $f(16) = 8$; $f(8) = 4$; $f(4) = 2$; $f(2) = 1$. So the sequence would be 6, 3, 10, 5, 16, 8, 4, 2, 1.

   I write the following code which allows the user to enter an integer n, and checks the Collatz conjecture for that integer n, by printing out the sequence of values which follow n until 1 comes out (e.g., if the user entered `6`, then the program should print out `3 10 5 16 8 4 2 1`).

```
// Definition of next_collatz() would go here.

int main() {
   cout << "Enter N: ";
   int N;
   cin >> N;
   while(N != 1){
      next_collatz(N); // PAY CLOSE ATTENTION TO THIS LINE
      cout << N << " ";
   }
}
```

   Write the function `next_collatz()` so that this code works. Pay close attention to how the function is called in the code above, and how that relates to the signature line of your function.

5. I have the following code, where I declare two arrays, and then try to copy one into the other:

```
int main() {
   int arr[10] = {4, 7, 3, 0, -1, 2, 8, 5, 1, 2};
   int arr2[10];
   arr2 = arr; // hmmm....
}
```

   Then I remember that `arr2 = arr;` is illegal in C++. D'oh!

   Sensing that copying arrays of `int`s is going to be a common operation in my code, I decide to write a function called `copy()`, which simplifies copying the entries of one array into another array.

   Write such a function `copy()`, and then replace the line `arr2 = arr;` with a line containing a call to this function, such that after that line, `arr2` and `arr` have equal entries.

   Remember that functions in C++ cannot return arrays; and that arrays can be parameters of functions, but that (important!) they are always automatically passed by *reference*.

6. So far, for all our programs, any input has come via `cin` statements. However, you can also write programs which take their inputs from the command line. That means, for example, if you give your executable program the name `prog`, then you can run the program from the terminal with input arguments `first` and `second` by the line

```
./prog first second
```

   Here's how to accomplish this. You add arguments to the `int main()` function:

```
int main(int argc, char *argv[]){
```

   Here, `argc` is the number of arguments used in the command line when you run the program, including the name of the program itself; and `argv[]` is an array holding the arguments themselves, where the first element is the name of the program itself. So, for example, if you ran

```
./prog first second
```

   then `argc` would be 3, and `argv` would be an array of strings of length 3, with `argv[1]` equaling `"first"`, `argv[2]` equaling `"second"`, and `argv[0]` being a string holding the full name of the program itself. (The boundaries between different arguments are determined by whitespace – so the arguments themselves can't contain spaces. Actually, you can get around that by enclosing arguments with spaces in quotation marks.)

Write a program that takes a `name` and a positive integer as command line arguments, and then creates many empty files with names of the form `name1.txt`, `name2.txt`, `name3.txt`, etc. For example, if I compile and name the executable `q6`, then if I run the program from the command line using

```
./q6 alice 5
```

then the program should create files named `alice1.txt`, `alice2.txt`, `alice3.txt`, `alice4.txt`, and `alice5.txt`, all of them containing no data. Your program should print *nothing, unless it is called with fewer than 3 command line arguments, in which case it should print a message notifying the user that there is an error.*

Some hints: review output filestreams from HW 1. Also, you'll probably need the functions `std::to_string()` and `std::stoi()`. The first of these functions converts an `int` to a string of digits, and the second function does the reverse, converting strings that look like integers into `int`s.

7. Write a recursive, loop-free function which receives a double `x` and an integer `n`, and returns x^n. Don't use the math library, either. You do not need to worry about the possibility of overflow (when your computed value is too large to fit in 8 bytes). But do be sure that your function accommodates **negative integer exponents**.

8. Write a recursive function called `log_floor()`, which receives an `int` value `n`, which may be assumed to be ≥ 1. The function should return the greatest integer `k` such that $2^k \leq n$. For example, `log_floor(31)` should return `4`, because $2^4 \leq 31$, but 2^5 is greater than 31.

   Your function should not use any loop, and employ recursion. Hint: `log_floor(n)` should generally be 1 greater than `log_floor(n/2)`.

9. Write a recursive function named `reverse_range()` which receives three arguments: an array of `int`s named `arr`, and two `int`s named `i` and `j` (with `i` assumed to be $\leq$ `j`). The function should return nothing, but it should reverse the order of the entries between index `i` and index `j`.

   For example, if an array was defined by

   ```
   int z[6] = {17, 29, 31, 14, 85, 60};
   ```

   then after `reverse_range(z, 1, 4)` was called, `z` would be {`17, 85, 14, 31, 29, 60`}, with the second to fifth entries reversed.

   Your recursive function should not employ any loops. Suggested strategy: have your function swap the values of `arr[i]` and `arr[j]`, and then reverse the values between `arr[i+1]` and `arr[j-1]`.