CISS245: Advanced Programming
Assignment 1

Name: _____

Q1. This is a practice on pointers and using the free store (memory heap).

Your are given the function `factorial()` which computes the factorial of n. (Don't remember factorial? Review right away. Also google "factorial" and read about the importance of this function.) The goal is to write a similar function `factorial2()` that does the same thing but in a different way.

The function `factorial()` is really old stuff from the early days of CISS240. It uses integer variable `i` to run over the integer from `1` to `n`. The product of these integers are stored in `p`. The function then returns a copy of the value of `p` which would be the factorial of `n` after the loop.

The function `factorial2()` does the same thing but uses pointers `i` and `p`. These pointers must point to integer values in the free store (or memory heap) and therefore you must allocate memory for them. Furthermore you must deallocate the memory used to prevent memory leaks. Because the memory (i.e., the final factorial value) used by `p` will be deallocate, you must retain it in an `int` variable. That's the purpose of `ret`.

To make `factorial()` and `factorial2()` as similar as possible, notice that I have mirrorred the `ret` variable in `factorial`. For `factorial`, the variable `ret` is redundant: obviously you could have returned the value of `p` in `factorial()`. I'm using `ret` in `factorial()` to make it easier for you when you're writing `factorial2()`. Note that for the case of `factorial2()`, you must make a copy of the `int` value from `*p` to `ret`. Why? Because after you deallocate the memory used by `p` (i.e., release the memory that `p` points to back to the free store), you cannot refer to `*p` anymore. As an aside, the more natural way to write the `factorial()` function is this (see CISS240 notes/textbook):

```
int factorial(int n)
{
    int p = 1;
    for (int i = 1; i <= n; i++)
    {
        p *= i;
    }
    return p;
}
```

The goal is to complete `factorial2()`. The following skeleton code must be used. Do NOT declare any extra variables in `factorial2()`. Do NOT change `main()`.

```
#include <iostream>

int factorial(int n)
{
    int p = 1;
    int i = 1;

    // Compute factorial of n and store in p
    for (i = 1; i <= n; i++)
    {
         p *= i;
    }

    int ret = p;
    return ret;
}

int factorial2(int n)
{
    // Declare pointer p, allocate memory for p, and initialize *p
    int * p = new int;
    *p = 1;

    // Declare pointer i, allocate memory for i, and initialize *i
    // TO BE COMPLETED.

    // Compute the factorial of n and store at *p
    // TO BE COMPLETED.

    int ret = *p; // Copy factorial stored at *p into ret
    // Deallocate memory for i. TO BE COMPLETED.
    // Deallocate memory for p. TO BE COMPLETED.
    return ret;
}


int main()
```

```
{
    int n = 0;
    std::cin >> n;
    std::cout << factorial(n) << ' '
              << factorial2(n) << '\n';
    return 0;
}
```

Test 1

```
4
24 24
```

Q2. This is a practice on using pointers and the free store (memory heap).

The goal is to write a function `pi()` (not the number `pi = 3.14159...`) such that `pi(x)` is the number of primes $\leq$ `x`. `x` is a `double`. We'll use the sieve of Eratosthenes. This was mentioned in CISS240. See next page for a review of the algorithm. Here's the skeleton code:

```
#include <iostream>

int pi(double x)
{
    int n = x;
    int * p;

    // Allocate an array of size n for p to point to.

    // Perform the sieve of Eratosthenes on the array that p points to.

    // Count the number of primes in the array that p points to and store
    // in ret.
    int ret;

    // Deallocate the memory used by p.

    return ret;
}

int main()
{
    double x;
    std::cin >> x;
    std::cout << pi(x) << '\n';
    return 0;
}
```

Test 1.

```
100
25
```

(There are 25 primes in 0, 1, 2, ..., 100.)

Test 2.

```
101.5
26
```

(There are 26 primes in 0, 1, 2, ..., 101.)

Sieve of Eratosthenes

The sieve of Eratosthenes is a method to compute primes up to a certain positive integer n. The following illustrates the idea.

Let's compute all the prime up to 99. Create an array `x` of size 100, i.e., you have `x[0]`, `x[1]`, `x[2]`, ... `x[99]`. After we're done with the computation, `x[i]` will be `1` if `i` is a prime. For instance when the computation ends, the first 7 values in `x` which are `1` are `x[2]`, `x[3]`, `x[5]`, `x[7]`, `x[11]`, `x[13]`, `x[17]`. Therefore from the array, you can tell that 2, 3, 5, 7, 11, 13, 17 are all primes. However `x[0]`, `x[1]`, `x[4]`, `x[6]`, `x[8]`, `x[9]`, `x[10]`, `x[12]`, `x[14]`, `x[15]`, `x[16]` and all `0` because 0, 1, 4, 6, 8, 9, 10, 12, 14, 15, 16 are not primes. By counting the number of `1`'s in `x`, you will then have the number of primes up to 99.

Here's the idea of the algorithm. First initialize `x[0]` and `x[1]` to `0` (since `0` and `1` are not primes) and all other values of `x` to `1`. Now we begin.

The first value of `x` that is `1` is at index `2`. Therefore `2` is a prime. Call it a killer prime. `2` will now kill off all the multiple of `2` except for `2`, i.e., you set `x[4]`, `x[6]`, `x[8]`, `x[10]`,... `x[98]` to `0`. Remember that you are at index `2`.

Now for the next stage, from index `2`, go to the next index `i` such that `x[i]` is `1`. This would be `x[3]`. So `3` is the next killer prime. `3` will kill of all the multiples of `3` except for `3`, i.e., you set `x[6]`, `x[9]`, `x[12]`, `x[15]`,..., `x[99]` to `0`. (Note that `x[6]` was already set to `0` by killer prime `2`. So `x[6]` is set to `0` twice.) Recall that you are at index `3`.

Now from index `3`, go to the next index `i` such that `x[i]` is `1`. This would be `x[5]`. So `5` is the next killer prime. `5` will kill of all the multiples of `5` except for 5, i.e., you set `x[10]`, `x[15]`, `x[20]`, `x[25]`,... `x[95]` to `0`.

Go to the next index `i` such that `x[i]` is `1`. This would be `x[7]`. So you set `x[14]`, `x[21]`, `x[28]`, `x[35]`,... `x[98]` to `0`.

If you keep doing this, you'll see that an index value `i` is a prime exactly when `x[i]` is `1`. By counting all such `i` values, you will then have all the primes from 0 to 99.

Another thing to note is that you don't have to locate the killer primes all the way up to 99. You can in fact stop the search for killer primes beyond $\sqrt{99} \approx 9.95$. For instance, if you continue algorithm above beyond `7`, the next killer prime would have been `11`. But the multiples of `11` greater than `11` are 22, 33, 44, 55, 66, 77, 88, and 99, and these are already killed by 2, 3, 2, 5, 2, 7, 2, 3.

(Extra challenge especially if you have taken discrete 1, MATH225, you should try to

prove that for the general case of performing Erathothenes sieve on n, the algorithm does find all the primes when the search for kill primes terminate at $\sqrt{n}$.)

You should trace the above on paper, then write down the pseudocode, and then translate into C++. You are also strongly advised to create more test cases.

(Instead of using an integer array that contains `0`'s and `1`'s, another option is to use an array of boolean values.)

Q3. This is a practice on passing pointers to functions.

We want to write some functions that performs "increment by a given amount". First run this progrm:

```
#include <iostream>

void inc_by_version1(int i, int amt)
{
    i += amt;
}

void inc_by_version2(int & i, int amt)
{
    i += amt;
}

int main()
{
    int i = 40;

    inc_by_version1(i, 2);
    std::cout << i << ' ';
    i = 40;

    inc_by_version2(i, 2);
    std::cout << i << ' ';
    i = 40;

    std::cout << '\n';
    return 0;
}
```

The function `inc_by_version1()` does not work: `i` in `main()` is not increment at all. That's because the in this function call, we are using pass by value.

The function `inc_by_version2()` does work: we're using pass by reference so that the `i` in `inc_by_version2()` is a reference to the `i` in `main()`. Review references if necessary.

Complete the following third version using the following given code; The third version uses a pointer as parameter:

```
#include <iostream>

void inc_by_version1(int i, int amt)
{
    i += amt;
}

void inc_by_version2(int & i, int amt)
{
    i += amt;
}

void inc_by_version3(int * i, int amt)
{
    // TO BE COMPLETED.
}

int main()
{
    int i = 40;

    inc_by_version1(i, 2);
    std::cout << i << ' ';
    i = 40;

    inc_by_version2(i, 2);
    std::cout << i << ' ';
    i = 40;

    inc_by_version3(&i, 2);
    std::cout << i << ' ';
    i = 40;

    std::cout << '\n';
    return 0;
}
```

Q4. Vectors are extremely important. A 2-dimensional vector in math looks like this

$$\langle 2, 3 \rangle$$

You can think a vector as representing motion. For the above vector, you can think of that vector as a motion of an object that moves by 2 units along the x-axis and by 3 units along the y-axis.

Clearly we can model the above vector $\langle 2, 3 \rangle$ in C++ using two variables with values 2 and 3 respectively.

```
int x = 2, y = 3;
std::cout << '<' << x << ", " << y << '>' << std::endl;
```

Vectors appear in many sciences. In particular, they appear in Physics. Therefore in CS they are extremely important in computer simulations of moving physical objects, whether you are talking about rockets, planes, or cars. And of course there are moving objects in games and so games use a lot of vectors. But it's more than that. Vectors are used to model light particles which allows computer software to render lighting and shadows of object simulations. In terms of careers, there's no doubt that if you want to work in space exploration (SpaceX, NASA) or self-driving cars (Tesla, Waymo, Mobileye) or aerospace/aviation (Lockheed-Martin, Boeing, Northrop Grumman) or computer graphics (NVIDIA, Sony, Industrial Light and Magic), you will need to know vectors.

One important operation on vectors is vector addition. Here's an example:

$$\langle 2, 3 \rangle + \langle 1, 4 \rangle = \langle 3, 7 \rangle = \langle 2 + 1, 3 + 4 \rangle$$

In other words, you just add the values according to their position or coordinate place. Here's another example:

$$\langle 5, -1 \rangle + \langle 2, 3 \rangle = \langle 7, 2 \rangle$$

Write a function `vec2d_add()` that adds two 2-dimensional vectors of integer values. I have provided a version that uses references for parameters. Note that the two versions have the same function name. That is perfectly fine for the compiler since their function signatures are different – the function name `vec2d_add` is overloaded.

```
#include <iostream>

// version of vec2d_add that uses references
void vec2d_add(int & x0, int & y0,
```

```
                int x1, int y1,
                int x2, int y2)
{
    x0 = x1 + x2;
    y0 = y1 + y2;
}


// version of vec2d_add that uses pointers
void vec2d_add(int * x0, int * y0,
                int x1, int y1,
                int x2, int y2)
{
    // TO BE COMPLETED.
}


void vec2d_println(int x, int y)
{
    std::cout << '<' << x << ", " << y << ">\n";
}


int main()
{
    int x0 = 0, y0 = 0, x1, y1, x2, y2;
    std::cin >> x1 >> y1 >> x2 >> y2;

    int old_x0 = x0, old_y0 = y0;

    vec2d_add(x0, y0, x1, y1, x2, y2);
    vec2d_println(x0, y0);

    x0 = old_x0;
    y0 = old_y0;

    vec2d_add(&x0, &y0, x1, y1, x2, y2);
    vec2d_println(x0, y0);

    return 0;
}
```

Test 1

```
1 2 3 4
<4, 6>
<4, 6>
```

Q5. The following is a practice on using pointers and arrays in the free store.

Because your cellphone/smartphone went off in your history class, your History prof has given you the following punishment.

You are sent off to a mansion with n doors.

- You run from door 0 to door $n-1$ and open all of them.
- Next, you run from door $n-1$ to door 0 and close every *other* door.
- Next, you run from door 0 to door $n-1$ opening every *third* door.
- Next, you run from door $n-1$ to 0, closing every *fourth* door.
- Etc. You stop if a run would only open or close one door.

At the end of the above process, you report to your History prof how many doors are open.

Here's an example. Suppose $n = 10$.

- You run from door 0 to door $n-1 = 9$ and open all of them, i.e., you open doors 0, 1, 2, 3, 4, 5, 6, 7, 8, 9.
- Then you close doors 9, 7, 5, 3, 1.
- Then you open doors 0, 3, 6, 9. (Note: Door 6 is already open. It's OK to open a door that is open.)
- Then you close doors 9, 5, 1.
- Then you open doors 0, 5.
- Then you close doors 9, 3.
- Then you open doors 0, 7.
- Then you close doors 9, 1.
- Then you open doors 0, 9.
- And you stop because at this stage if you want to close doors, you can only close door 9.

(The above tells you what I have always mentioned to you: concrete examples are best for understanding general/abstract statements.)

Besides programming, you now know something about the dangers of leaving your smartphone on, right? The above won't happen in my classes since the punishment is to buy donuts for everyone for the next class.

Complete the following program. Note that you must use the skeleton code.

```
int opendoors(int n)
{
    bool * open;   // open[i] is true when door i is open.
    int count = 0; // This will count the number of values in the
                   // array that open points to which are true.

    // Allocate an array of n values for open to point to.

    // Scan (left and right) and update the array that open points to
    // a certain number of times.

    // Update count.

    // Deallocate the memory used by the open pointer.

    return count;
}


int main()
{
    int n;
    std::cin >> n;
    std::cout << opendoors(n) << std::endl;
    return 0;
}
```

You are very strongly advised to do several test cases by hand and use the test cases to help you check your program. You might want to check with others in the class on your test data. Of course passing some tests does not necessarily mean your program is absolutely correct. But the more tests you do the greater the likelihood that your program might be correct.

Note. Extra DIY challenge. Suppose I allow you to not use the skeleton code. Can you write a function that is faster? This problem has been a 245 assignment question for at least 10 years. I think only one student mentioned to me that something is very fishy about this problem. Hidden in this problem is something very deep in math. With lots of experiments, you might see a pattern. But the question is: Does the pattern really hold for all n? And why?

Q6. You already know that C++ performs character, integer and double inputs like this:

```
char c = ' ';
int i = 0;
double d = 0;
std::cin >> c;
std::cin >> i;
std::cin >> d;
// or:
// std::cin >> c >> i >> d;
```

The father of C++, i.e. the C programming language, performs character, integer, and double input like this:

```
#include <cstdio>

int main()
{
    char c = ' ';
    int i = 0;
    double d = 0;

    scanf("%c", &c);  // char input for c
    scanf("%d", &i);  // int input for n
    scanf("%lf", &d); // double input for d
    std::cout << c << ' ' << i << ' ' << d << '\n';

    return 0;
}
```

You can compile that using a C++ compiler. But if you use a C compiler you have to change `cstdio` to `stdio.h`. C also prints things in a different way:

```
#include <cstdio>

int main()
{
    char c = '#';
    int i = 0;
    double d = 0;
    char s[1024] = "hello world";

    printf("%c\n", c);
    printf("%d\n", i);
    printf("%lf\n", d);
    printf("%s\n", s);

    return 0;
}
```

The purpose of this question is to write a series of input functions similar to `scanf`. Here's the skeleton code that you MUST use.

```
#include <iostream>

void scanf(int *);
void scanf(char *);
void scanf(double *);

int main()
{
    char c = 0;
    int i = 0;
    double d = 0;
    scanf(&c);     // same effect as std::cin >> c
    scanf(&i);     // same effect as std::cin >> i
    scanf(&d);     // same effect as std::cin >> d
    std::cout << c << std::endl;
    std::cout << i << std::endl;
    std::cout << d << std::endl;

    return 0;
}

// Implement the functions here
```

Your goal is to implement the given three function prototypes.

Test 1

```
A 42 3.14
A
42
3.14
```

Q7.

This is a practice on pointers and using the free store (memory heap).

In the area of computer vision, it's very common to perform certain "weighted averaging" computation on 2D arrays of pixels. For instance suppose an array has the following value

```
0 0 1 0 1 3 5 1
```

For the value at index 2,

```
0 0 1 0 1 3 5 1
```

if you compute the average of the value at index 1, 2, 3, you have `(0 + 1 + 0)/3` which will give you `0` (I'm doing integer division here):

```
0 0 1 0 1 3 5 1
    0
```

For the value at index 5,

```
0 0 1 0 1 3 5 1
```

¡¡¡¡¡¡¡ HEAD if you compute the average of the value at index 5, 6, 7, you have ======= if you compute the average of the value at index 4, 5, 6, you have ¿¿¿¿¿¿¿ fd78148a635584414d0f8a155afb8fb53d60ce54 `(1 + 3 + 5)/3` which will give you `3`:

```
0 0 1 0 1 3 5 1
        3
```

For this computation the average must use the values from the given array. So you'll need to put the average values into another array. Note that the first value and the last value do not have left and right neighbors, so the computation is not performed on the first and last value – they are just copied to the target array. I'll then have these two arrays where the top is the given array and the bottom is the averaged array:

```
0 0 1 0 1 3 5 1
0 0 0 0 1 3 3 1
```

This is an oversimplified example of what is called a linear filter. Linear filters are part of an area of study called signal processing. Computer vision intersects signal processing.

You can generalize the above a little. Let me do this in 2 steps.

First, the average computation

```
(0 + 1 + 0)/3
```

can be generalized so that "weights" `a`, `b`, `c`, are added to the above computation:

```
(a * 0 + b * 1 + c * 0)/(a + b + c)
```

For the example above, the weights are $\texttt{a} = 1, \texttt{b} = 1, \texttt{c} = 1$. If I use weights $1, 2, 1$, then at index 2,

```
0 0 1 0 1 3 5 1
    0
```

the weighted average for `0 1 0` is

```
(1 * 0 + 2 * 1 + 1 * 0) / (1 + 2 + 1) = 2 / 4
```

which is `0` if I use integer division.

Second, you can further generalize the above by averaging over different number of values. In the above example, a value is averaged over three values. Now let me average over 5 values. For the value at index 2, I can average over 5 values with weights $1, 4, 7, 4, 1$. For the value at index 2:

```
0 0 1 0 1 3 5 1
```

I then get

```
1 * 0 + 4 * 0 + 7 * 1 + 4 * 0 + 1 * 1 / (1 + 4 + 7 + 4 + 1) = 9 / 17
```

which is 0 if I use integer division.

```
0 0 1 0 1 3 5 1
    0
```

In this case when there are five weights, the first two values and the last two values are not averaged – we just copy them to the target array.

```
#include <iostream>

void filter(int x[], int n)
{
    int * p;
```

```
    // Allocate an integer array of size n for p to point to.

    // Compute the weighted average values of x and store in the array
    // that p points to.

    // Copy the values in the array that p points to into
    // the values of array x.

    // Deallocate the memory used by p.

    return;
}


int main()
{
    int x[1024];

    int n = 0;
    std::cin >> n;
    for (int i = 0; i < n; ++i)
    {
        std::cin >> x[i];
    }
    filter(x, n);
    for (int i = 0; i < n; ++i)
    {
        std::cout << x[i] << ' ';
    }
    std::cout << '\n';

    return 0;
}
```

Test 1

```
8
0 0 1 0 1 3 5 1
0 0 0 0 1 3 3 1
```