Homework #3 (30 points)
Due: 7 April 11:59 pm

For this problem set, you should be able to put all your code for each section into a single source/text file (though you may have to comment out earlier parts of your solution to test later parts). Clearly mark each subsection with comments, and submit a .zip file containing all your source/text files.

1 Additional Material

1.1 Functions

1.1.1 Default Arguments

Say you have a function with 1 argument, but that argument is usually the same. For instance, say we want a function that prints a message n times, but most of the time it will only need to print it once:

```
1 void printNTimes(char *msg, int n) {
2     for( int i = 0; i < n; ++i) {
3         cout << msg;
4     }
5 }
```

Rather than writing `printNTimes("Some message", 1);` every time, C++ allows *default arguments* to be defined for a function:

```
1 void printNTimes(char *msg, int n = 1) {
2     for( int i = 0; i < n; ++i) {
3         cout << msg;
4     }
5 }
```

Declaring the function argument as `int n = 1` allows us to call the function with `printNTimes("Some message");`. The compiler automatically inserts 1 as the second argument.

You may have multiple default arguments for a function:

```
1 void printNTimes(char *msg = "\n", int n = 1) {
2     for( int i = 0; i < n; ++i) {
3         cout << msg;
```

```
    }
}
```

Now, to print one newline, we can simply write `printNTimes();`. However, C++ does not allow skipping arguments, so we could not print k newlines by writing `printNTimes(k);`. To do that, we'd need to say `printNTimes("\n", k);`.

1.1.2 Constant Arguments

It's often useful to specify that arguments to a function should be treated as constants. As with regular variables, we can declare function arguments to be `const`:

```
void print(const int n) {
  cout << n;
}
```

This is particularly useful when we are passing values by reference to a function, but don't want to allow the function to make any changes to the original value:

```
void print(const long &x) { // Pass-by-reference avoids overhead
                            // of copying large number
    cout << x;
}

int main() {
    long x = 234923592;
    print(x); // We are guaranteed that x
              // will not be changed by this
    return 0;
}
```

In general, if you know a value shouldn't be changing (particularly a function argument), you should declare it `const`. That way, the compiler can catch you if you messed up and tried to change it somewhere.

1.1.3 Random Number Generation Functions

The C++ standard libraries include the `rand()` function for generating random numbers between 0 and `RAND_MAX` (an integer constant defined by the compiler). These numbers are not truly random; they are a random-seeming but deterministic sequence based on a particular "seed" number. To make sure we don't keep getting the same random-number sequence, we generally use the current time as the seed number. Here is an example of how this is done:

```
#include <iostream>
#include <cstdlib> // C standard library -
                   // defines rand(), srand(), RAND_MAX
```

```
#include <ctime>   // C time functions - defines time()
int main() {
    srand( time(0) ); // Set the seed;
                      // time(0) returns current time as a number
    int randNum = rand();
    std::cout << "A random number: " << randNum << endl;
    return 0;
}
```

1.2 Pointers

1.2.1 Pointers to Pointers

We can have pointers to any type, including pointers to pointers. This is commonly used in C (and less commonly in C++) to allow functions to set the values of pointers in their calling functions. For example:

```
void setString(char **strPtr) {
    int x;
    cin >> x;
    if(x < 0)
        *strPtr = "Negative!";
    else
        *strPtr = "Nonnegative!";
}

int main() {
    char *str;
    setString(&str);
    cout << str; // String has been set by setString
    return 0;
}
```

1.2.2 Returning Pointers

When you declare a local variable within a function, that variable goes out of scope when the function exits: the memory allocated to it is reclaimed by the operating system, and anything that was stored in that memory may be cleared. It therefore usually generates a runtime error to return a pointer to a local variable:

```
int *getRandNumPtr() {
    int x = rand();
    return &x;
}

```

```
6 int main() {
7     int *randNumPtr = getRandNumPtr();
8     cout << *randNumPtr; // ERROR
9     return 0;
10 }
```

Line 8 will likely crash the program or print a strange value, since it is trying to access memory that is no longer in use – `x` from `getRandNumPtr` has been deallocated.

1.3 Arrays and Pointers

1.3.1 Arrays of Pointers

Arrays can contain any type of value, including pointers. One common application of this is arrays of strings, i.e., arrays of `char *`'s. For instance:

```
1 const char *suitNames[] = {"Clubs", "Diamonds", "Spades", "Clubs"};
2 cout << "Enter a suit number (1-4): ";
3 unsigned int suitNum;
4 cin >> suitNum;
5 if(suitNum <= 3)
6     cout << suitNames[suitNum - 1];
```

1.3.2 Pointers to Array Elements

It is important to note that arrays in C++ are pointers to continuous regions in memory. Therefore the following code is valid:

```
1 const int ARRAY_LEN = 100;
2 int arr[ARRAY_LEN];
3 int *p = arr;
4 int *q = &arr[0];
```

Now `p` and `q` point to exactly the same location as `arr` (ie. `arr[0]`), and `p`, `q` and `arr` can be used interchangeably. You can also make a pointer to some element in the middle of an array (similarly to `q`):

```
1 int *z = &arr[10];
```

1.4 Global Scope

We discussed in lecture how variables can be declared at *global scope* or *file scope* – if a variable is declared outside of any function, it can be used anywhere in the file. For anything besides global constants such as error codes or fixed array sizes, this is usually a bad idea; if you need to access the same variable from multiple functions, most often you should simply

pass the variable around as an argument between the functions. Avoid global variables when you can.

2 A Simple Function

What would the following program print out? (Answer without using a computer.)

```
1  void f(const int a = 5)
2  {
3      std::cout << a*2 << "\n";
4  }
5
6  int a = 123;
7  int main()
8  {
9      f(1);
10     f(a);
11     int b = 3;
12     f(b);
13     int a = 4;
14     f(a);
15     f();
16 }
```

3 Fix the Function

Identify the errors in the following programs, and explain how you would correct them to make them do what they were apparently meant to do.

3.1

```
1 #include <iostream>
2
3 int main() {
4     printNum(35);
5     return 0;
6 }
7
8 void printNum(int number) { std::cout << number; }
```

(Give two ways to fix this code.)

3.2

```
#include <iostream>

void printNum() { std::cout << number; };

int main() {
    int number = 35;
    printNum(number);
    return 0;
}
```

(Give two ways to fix this code. Indicate which is preferable and why.)

3.3

```
#include <iostream>

void doubleNumber(int num) {num = num * 2;}

int main() {
    int num = 35;
    doubleNumber(num);
    std::cout << num; // Should print 70
    return 0;
}
```

(Changing the return type of `doubleNumber` is not a valid solution.)

3.4

```
#include <iostream>
#include <cstdlib> // contains some math functions

int difference(const int x, const int y) {
    int diff = abs(x - y); // abs(n) returns absolute value of n
}

int main() {
    std::cout << difference(24, 1238);
    return 0;
}
```

3.5

```
#include <iostream>

int sum(const int x, const int y) {
    return x + y;
}

int main() {
    std::cout << sum(1, 2, 3); // Should print 6
    return 0;
}
```

3.6

```
#include <iostream>
const int ARRAY_LEN = 10;

int main() {
    int arr[ARRAY_LEN] = {10}; // Note implicit initialization of
                               // other elements
    int *xPtr = arr, yPtr = arr + ARRAY_LEN - 1;
    std::cout << *xPtr << ' ' << *yPtr; // Should output 10 0
    return 0;
}
```

4 Sums

Make sure to use `const` arguments where appropriate throughout this problem (and all the others).

4.1

Write a single `sum` function that returns the sum of two integers. Also write the equivalent function for taking the sum of two `doubles`.

4.2

Explain why, given your functions from part 1, `sum(1, 10.0)` is a syntax error. *(Hint:* Think about promotion and demotion – the conversion of arguments between types in a function call. Remember that the compiler converts between numerical types for you if necessary.) [1 point]

4.3

Write 2 more functions such that you can find the sum of anywhere between 2 and 4 integers by writing `sum(num1, num2, ...)`.

4.4

Now write just one function that, using default arguments, allows you to take the sum of anywhere between 2 and 4 integers. What would happen if you put both this definition and your 3-argument function from part 3 into the same file, and called `sum(3, 5, 7)`? Why?

4.5

Write a single `sum` function capable of handling an arbitrary number of integers. It should take two arguments, include a loop, and return an integer. *(Hint: What data types can you use to represent an arbitrarily large set of integers in two arguments?)*

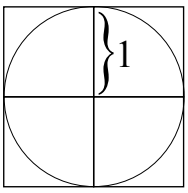
4.6

Now rewrite your function from 4.5 to use recursion instead of a loop. The function signature should not change. Thinking about pointer arithmetic may help you.

5 Calculating π

This problem is a bit tricky, but it's a good exercise in writing a program that actually does something neat. It will also familiarize you with using random numbers.

Using a "Monte Carlo" method – that is, a randomized simulation – we can compute a good approximation of π. Consider a circle of radius 1, centered on the origin and circumscribed by a square, like so:



Imagine that this is a dartboard and that you are tossing darts at it randomly. With enough darts, the ratio of darts in the circle to total darts thrown should be the ratio between the area of the circle (call it a) and the area of the square (4): $\frac{\text{total darts}}{\text{darts in circle}} = \frac{4}{a}$. We can use this ratio to calculate a, from which we can then find $\pi = \frac{a}{r^2}$.

We can simplify the math by only considering the first quadrant, calculating the ratio of the top right square's area to the area of the single quadrant. Thus, we will actually find $\frac{a}{4}$, and then compute $\pi = 4 \times \frac{\frac{a}{4}}{r^2}$.

We'll build a function step by step to do all this.

5.1

Define variables to store the x and y coordinates of a particular dart throw. Initialize them to random `doubles` in the range [0, 1] (simulating one dart throw). *(Hint: remember that* ***`rand()`*** *returns a value in the range [0, RAND_MAX]; we just want to convert that value to some value in [0, 1].)*

5.2

Place your x and y declarations in a loop to simulate multiple dart throws. Assume you have a variable `n` indicating how many throws to simulate. Maintain a count (declared outside the loop) of how many darts have ended up inside the circle. (You can check whether a dart is within a given radius with the Euclidean distance formula, $d^2 = x^2 + y^2$; you may find the `sqrt` function from the `<cmath>` header useful.)

5.3

Now use your loop to build a π-calculating function. The function should take one argument specifying the number of dart throws to run (`n` from part 2). It should return the decimal value of pi, using the technique outlined above. Be sure to name your function appropriately. Don't forget to initialize the random number generator with a seed. You should get pretty good results for around 5,000,000 dart throws.

6 Array Operations

6.1

Write a function `printArray` to print the contents of an integer array with the string `", "` between elements (but not after the last element). Your function should return nothing.

6.2

Write a `reverse` function that takes an integer array and its length as arguments. Your function should reverse the contents of the array, leaving the reversed values in the original array, and return nothing.

6.3

Assume the existence of two constants `WIDTH` and `LENGTH`. Write a function with the following signature:

```
void transpose(const int input[][LENGTH], int output[][WIDTH]);
```

Your function should transpose the `WIDTH` × `LENGTH` matrix in `input`, placing the `LENGTH` × `WIDTH` transposed matrix into `output`. (See http://en.wikipedia.org/wiki/Transpose#Examples for examples of what it means to transpose a matrix.)

6.4

What would happen if, instead of having `output` be an "out argument," we simply declared a new array within `transpose` and returned that array?

6.5

Rewrite your function from part 2 to use pointer-offset notation instead of array-subscript notation.

7 Pointers and Strings

7.1

Write a function that returns the length of a string (`char *`), excluding the final NULL character. It should not use any standard-library functions. You may use arithmetic and dereference operators, but not the indexing operator (`[]`).

7.2

Write a function that swaps two integer values using call-by-reference.

7.3

Rewrite your function from part 2 to use pointers instead of references.

7.4

Write a function similar to the one in part 3, but instead of swapping two values, it swaps two pointers to point to each other's values. Your function should work correctly for the following example invocation:

```
1 int x = 5, y = 6;
2 int *ptr1 = &x, *ptr2 = &y;
3 swap(&ptr1, &ptr2);
4 cout << *ptr1 << ' ' << *ptr2; // Prints "6 5"
```

7.5

Assume that the following variable declaration has already been made:

```
1 char *oddOrEven = "Never odd or even";
```

Write a single statement to accomplish each of the following tasks (assuming for each one that the previous ones have already been run). Make sure you understand what happens in each of them.

1. Create a pointer to a `char` value named `nthCharPtr` pointing to the 6th character of `oddOrEven` (remember that the first item has index 0). Use the indexing operator.
2. Using pointer arithmetic, update `nthCharPtr` to point to the 4th character in `oddOrEven`.
3. Print the value currently pointed to by `nthCharPtr`.
4. Create a new pointer to a pointer (a `char **`) named `pointerPtr` that points to `nthCharPtr`.
5. Print the value stored in `pointerPtr`.
6. Using `pointerPtr`, print the value pointed to by `nthCharPtr`.
7. Update `nthCharPtr` to point to the next character in `oddOrEven` (i.e. one character past the location it currently points to).
8. Using pointer arithmetic, print out how far away from the character currently pointed to by `nthCharPtr` is from the start of the string.