MECHTRON 2MD3 | Assignment 1

Question 1

> Declare A to be a pointer to integer and assign a value of 21 to its referent. How would you write an expression whose value is twice the value of A's referent?

```
#include <iostream>

using namespace std;
int main(){
    int b;
    int* a = &b; //a pointer to an integer
    *a = 21; //assigns a value of 21 to its referent
    int c = 2*(*a); // an expression whose value is twice the value of A's referent
    // or you could do:
    int d = 2*b;
    cout << c << endl;
    cout << d << endl;
}
```

Question 2

> Consider the following attempt to allocate a 10-element array of pointers to doubles and initialize the associated double values to 0.0. Rewrite the following (incorrect) code to do this correctly. (Hint: Storage for the doubles needs to be allocated.)

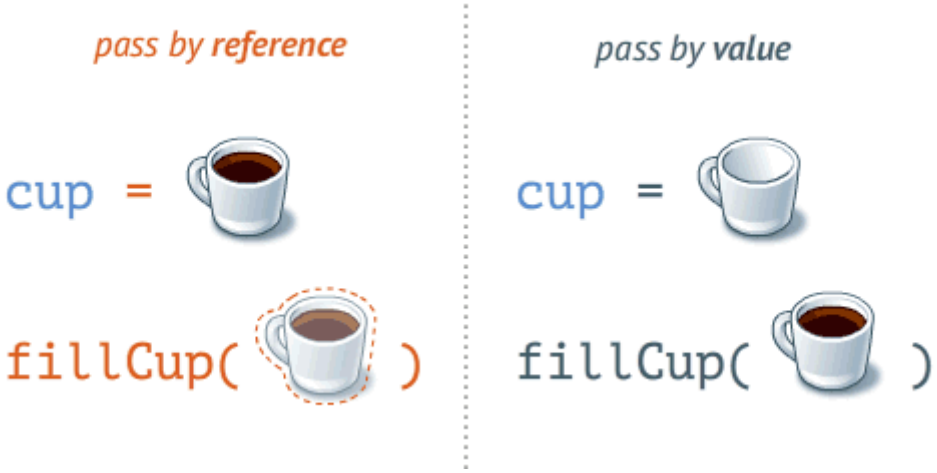
```
double* dp[10];
for (int i = 0; i < 10; i++) dp[i] = 0.0;
```

```
#include <iostream>
using namespace std;
int main(){
    double *dp = new double[10];
    for (int i = 0; i < 10; i++) dp[i] = 0.0;
    for(int i = 0; i<10; i++){
        cout<< dp[i] << "\t";
    }
}
```

Question 3

> What (if anything) is different about the behaviour of the following two function f and g that increment a variable and print its value?

The difference between the two functions `f` and `g` is the function f passes the function by value while function g passes the function by reference. The difference in function between the two pieces of code, is that for function `f` a local variable called x would be incremented by one while the original value that was passed remains unchanged. While for function `g` the variable that was inserted into the constructor where function `g` was called would be incremented by one.



Question 4

> White a short C++ function that takes a positive double value x and returns the number of times we can divide x by 2 before we get a number less than two.

```
double countUnderTwo(double x , int count){
    while(x>=2){
        x/=2;
        count++;
        if (count>10000){ //to prevent weird numbers
            return -1;
        }
    }
    return count;
}
```

Question 5

> The greatest common divisor, or GCD, of two positive integers n and m is the largest number j, such that n and m are both multiples of j. Euclid proposed a simple algorithm for computing GCD(n, m), where n > m, which is based on a concept known as the Chinese Remainder Theorem. The main idea of the algorithm is to repeatedly perform modulo computations of consecutive pairs of the sequence that starts (n, m,...), until reaching zero. The last nonzero number in this sequence is the GCD of n and m. For example, for n = 80,844 and m = 25,320, the sequence is as follows:
> So, GCD of 80,844 and 25,320 is 12. Write a short C++ function to compute GCD(n, m) for two integers n and m.

```
int GCD(int n , int m){
    while(m>0){
        int temp = m;
        m = n%m;
        n = temp;
        cout << "["<< n <<"]["<< m << "]" << endl;
    }
    return n;
}
```

Question 6

> The birthday paradox says that the probability that two people in a room will have the same birthday is more than half as long as the number of people in the room (n), is greater than 23. This property is not really a paradox, but many people find it surprising. Design a C++ program that can test this paradox by a series of experiments on randomly generated birthdays, which test this paradox for n = 5,10,15,20,...,100. You should run at least 100 experiments for each value of n. Your program should output a single comma-separated line for each n showing: 1) the value of n; 2) the number of experiments that returned two people in that test having the same birthday; 3) the measured probability of 2 people in the group having the same birthday. To calculate “measured probability” for each n: let c be the number of experiments in which at least 2 people had the same birthday and let e be the number of experiments. We define the “measured probability” as c/e. Example output is as follows:

```
#include <iostream>
#include <cstdlib>
#include <vector>
#include <time.h>

int experiment(int n){
    std::vector<int> list;
    srand(time(NULL));
    int count =0;
    bool check = false;
    for(int i =0;i<100;i++){
        for(int j =0;j<n;j++){
            list.push_back(1 + (rand() % 365));
        }
        for(int k=0;k<n;k++){
            for(int l=k+1;l<n;l++){
                if(list[k]==list[l]){
                    check = true;
                }
            }
        }
        if(check== true){
            count++;
        }
        list.clear();
        check = false;
    }
    return count;
}

double probability(int num){
    int e = 100;
    double measurement = num/e;
    return measurement;
}

int main(){
    int n = 10;
    int counter = experiment(n);
    double prob = (double)counter/100;
    std::cout<< n << "," << counter << "," << prob << std::endl;

}
```

Question 7

> Suppose we have a variable p that is declared to be a pointer to an object of type Progression using the classes of Section 2.2.3. Suppose further that p actually points to an instance of the

> class GeomProgression that was created with the default constructor. If we cast p to a pointer of type Progression and call p->nextValue(), what will be returned? Why? Please assume that the nextValue() function is public.

Lets say that we have the following code in main:

```
int main(){
    Progression* p; //a variable p that is declared to be a pointer to an object of type progres
    int n = 2;
    p = new GeomProgression(2); //p points to an instance of the class GeomProgression
    p->nextValue(); //cast p and call p->nextValue()
}
```

The value that would be returned in the `nextValue()` function would be the next value of the geometric sequence. In this case, since the geometric sequence would always start at 1. The next value would just be the value of `n` . If `n` is not declared the function automatically uses a value of `n=2` .

Question 8

> Write a short C++ program that creates a Pair class that can store two objects declared as generic types. Demonstrate this program by creating and printing Pair objects that contain five different kinds of pairs, such as `<int,string>` and `<float,long>` . Your class should include a print function to display pairs in the format “ `<value1, value2>` ”. An example main method testing 3 pairs and its associated output are below:

```cpp
#include <iostream>

template <typename t1, typename t2>

class Pair{
    public:
        t1 num1;
        t2 num2;

        Pair(t1 num1, t2 num2) : num1(num1), num2(num2) {}

        void print(){
            std::cout << "<" << num1 << ", " << num2 << ">" << std::endl;
        }
};

int main(){


    Pair<int, std::string> p1(1, "7.3");
    p1.print();

    Pair<std::string, double> p2("hello", 7.7);
    p2.print();

    Pair<float, long> p3(1.2, 777777773);
    p3.print();

   return 0;

}
```

Question 9

> Write a C++ class that is derived from the Progression class to produce a progression where each value is the absolute value of the difference between the previous two values. You can use this example from the 2md3_2023 git repo as a starting point:
>
> `2md3_2023/lecture_demos_ch02/polymorphism_demo.cpp`
>
> You should include a default constructor that starts with 2 and 200 as the first two values and a parametric constructor that starts with a specified pair of numbers as the first two values. Include a main method that tests your class using both constructors and generating a progression of 10 values for each. Your program should output 4 lines like the following:

```
class AProg : public Progression {
    public:
        AProg(long f = 2, long s = 200);  // constructor
    protected:
        virtual long firstValue();  // reset
        virtual long nextValue();   // advance
    protected:
        long second;  // second value
        long prev;

};

AProg::AProg(long f, long s)
    : Progression(f), second(s), prev(first) {}

long AProg::firstValue() {
  cur = second;
  prev = first;
  return cur;
}

long AProg::nextValue() {  // advance
  long temp = prev;
  prev = cur;
  cur -= temp;
  cur = abs(cur);
  return cur;
}

/** Test program for the progression classes */
int main() {
  Progression* p;
  cout << "Absolute progression with default constructor:" << endl;
  p = new AProg();
  p->printProgression(10);
  cout << "Absolute progression with custom constructor:" << endl;
  p = new AProg(4, 400);
  p->printProgression(10);
  return 0;
}
```