Idries Eagle-Masuak SN: 6868288

Part A:

1) There are two types of overloading:
    - Function Overloading
        - Overloading functions improves code readability by using the same name for the same activity. E.g.

          myFunction(); // Function that has no arguments
          myFunction(int, double, string); // Function that has 3 arguments
          myFunction(int *a); // Function that takes 1 argument

    - Operator Overloading
        - Allows you to reinterpret existing operator functionality and give it new meaning. E.g.

          int intNum = 2;
          double doubleNum = 3.5, average;
          average = (intNum + doubleNum) / 2;

2) A functor, or function object, is any type that implements the operator ().
This operator is also referred to as a call operator or an application operator. The C++ Standard Library primarily employs function objects as sorting criteria for containers and algorithms.

```
class mainClass
{
    public:
         double operator()(double a, double b)
         {
             return a < b;
          }
};
int main()
{
     mainClass c;
     double a = 16;
     double b = 55;
     double op = c(a, b);
}
```

3) A variable that holds the memory address of another variable is known as a pointer. After it is declared, a pointer can be initialized to any value. A* must be used to dereference pointers. A pointer can be redirected to any variable of the same type. A NULL value can be assigned to a pointer.

Idries Eagle-Masuak SN: 6868288

A reference is an alias for a variable that already exists. When a reference is declared, it must be initialized. NULL references are not permitted. Once a reference is initialized it cannot be changed.

4)

```
#include <iostream>

using namespace std;

int main()
{
    int n;
    cin >> n;
    int array[n];
    for (int i = 0; i < n; i ++)
    {
        array[i] = n;
    }
    cout << array << endl;
}
```

5) **my_function()** doesn’t have a memory leak, as unique_ptr will delete itself once the object is out of scope.

6) There could be a problem as **func** is a function reference and is referencing and returning the local variable **temp**

7)

```
struct Fraction {
    int numerator;
    int denominator;
    Fraction add(Fraction f1, Fraction f2){
        Fraction newFraction;
        newFraction.numerator = ((f1.numerator * f2.denominator) + (f1.denominator *
f2.numerator));
        newFraction.denominator = (f1.denominator * f2.denominator);
    }
};
```

8) Auto is used to iterate over the class types because they have member methods that determine the start and end. We don't have to declare the starting and ending indices for going through an array because auto does it for us. We get an error because arrays are not class types. Because vectors

are class types, we can use them alongside auto.

9)

```
float square(float width, float length=10.0) // the original
square function
{
return width*length;
}
// a missing function declaration for fp here
float *fp;
// to use fp, we do:
fp=&square; // use the function pointer
cout<<"The square area is "<<fp(1.0, 2.0)<<'\n'; //this output
should be 2.
```

10) The byte size of ptr1 is 20, as integers are 4 bytes, and there are 5 integers in the array, so 4*5. The byte size of ptr2 is 8 bytes as it is pointing to an array

11) The extension .h refers to the header file that defines the interfaces to various operating system components. We may incorporate them into the code to give definitions and declarations, as well as system calls and libraries. The header file provides C declarations and macro definitions that many source files can use. Using the C preprocessor command '#include,' you can request that a header file be included in your code.

The system header files define the interfaces to the various components of the operating system. They are used in the code to provide the definitions and declarations needed to call system functions and libraries. Declarations for interfaces between your program's source files are included in your own header files. It is a good idea to construct a header file for a collection of related declarations and macro definitions that are required in numerous different source files.

Inclusion of a header file has the same effect as including it in each source file that requires it. This method of copying would be time-consuming and error-prone. In a header file, the relevant declarations appear only once. If they need to be changed, they may be done in a single place, and applications that include the header file will use the new version when they are recompiled. The header file decreases the work of detecting and changing all the duplicates, as well as the risk that failing to recognize even one duplication will cause software difficulties.

12)

```
template<typename T>
T findMinArr(T* arr, int s)
{
    T min;

    min = arr[0];

    for(int i = 0; i < s; i++)
    {
        if(arr[i] < min)
        {
            min = arr[i];
        }
    }
    return min;
}
```

Idries Eagle-Masuak SN: 6868288

Part B:

1)

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

void level(int grade[], int size, int& honor, int& nonhonor){
    for (int i = 0; i < size; i++){
        if (grade[i] > 75){
            honor++;
        }
        else{
            nonhonor++;
        }
    }
}

int main()
{
    int array[6][4]={
        {60,80,90,75},
        {75,85,65,77},
        {80,88,90,98},
        {89,100,78,81},
        {62,68,69,75},
        {85,85,77,91}
        };
    int honor=0, nonhonor =0;
    int student=6;
    int gradesize=4;
    for(int i = 0; i < student; i++)
        level(array[i], gradesize, honor, nonhonor);

    cout <<"number of honor is " << honor << endl;
    cout <<"number of nonhonor is " << nonhonor << endl;
}
```

2) Output:

```
Base1's constructor called
Base2's constructor called
Derived's constructor called
```

Idries Eagle-Masuak SN: 6868288

Methods the *Derived* class have direct access to variables *n1B1* and *n2B1* in *Base1* as *Derived* inherits *Base1*

3)

```
#include<iostream> // Include
using namespace std; // Using

void swap(int &x1,int &x2) // Pointer
{
    int t;
    t=x1;
    x1=x2;
    x2=t;
}

void order(int &a,int &b) // Pointer
{
    if(a>b){
        swap(a,b);
    }
}

int main() // int, not Int
{
    int x,y;
    cout<<"two inputs?\n";
    cin>>x>>y;
    cout<<"before ordering: x="<<x<<",y="<<y<<endl;
    order(x,y);
    cout<<"after ordering: x="<<x<<",y="<<y<<endl; // Semicolon
}
```

4)

```
Test A; // Creates one Test object.
Test B = A; // Creates another test object which is a copy of A.
testOne(A); // Does nothing but print out “Testing”.
testTwo(B); // Does nothing but print out “More testing”.

Total number of Test objects: 2
```

Idries Eagle-Masuak SN: 6868288

5)

```
#include<iostream>
#include<string>
using namespace std;

class Cat {
    private:
        string name;
        int age;
    public:
        Cat() = default;
        ~Cat();
        Cat(string name, int age);
        Cat(const Cat & ); //copy constructor
        Cat(Cat && mCat);
};

Cat::~Cat() {
    cout << "Destructor" << endl;
}

Cat::Cat(string _name, int _age) {
    name = _name;
    age = _age;
    cout << "Cat constructing" << endl;
}

Cat::Cat(const Cat & copyCat) {
    name = copyCat.name;
    age = copyCat.age;
    cout << "Copy Cat constructor" << endl;
}

Cat::Cat(Cat && mCat) {
    name = move(mCat.name);
    age = move(mCat.age);
    cout << "Move Cat constructor" << endl;
}
int main() {
    Cat myCat("Daisy", 2);
    Cat secondCat = myCat;
    Cat thirdCat = move(secondCat);
    return 0;
}
```

Idries Eagle-Masuak SN: 6868288

6)

The print() method is added for testing purposes.
To merge the 4 constructors into 1, I set default values for each variables passed through the constructor

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

class Ship {
    private:
        int numberofstaff;
        double length;
        char purpose;
    public:
        Ship(int,double,char);
        void print();
};

Ship::Ship(int n = 15, double l = 14.1, char p = 'r'){
    numberofstaff = n;
    length = l;
    purpose = p;
}

void Ship::print(){
    cout << "Staff: " << numberofstaff << endl;
    cout << "Length: " << length << endl;
    cout << "Purpose: " << purpose << endl << endl;
}

int main(){
    Ship one = Ship();
    Ship two = Ship(2);
    Ship three = Ship(2, 3.5);
    Ship four = Ship(2, 3.5, 'a');

    one.print();
    two.print();
    three.print();
    four.print();
}
```

Idries Eagle-Masuak SN: 6868288

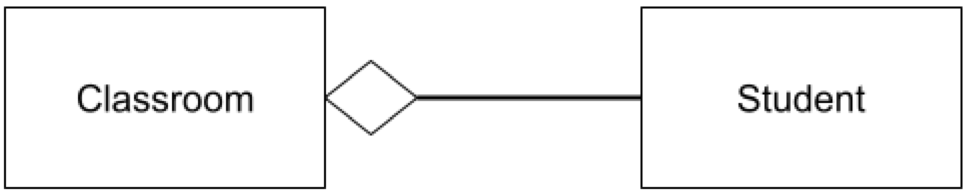
7) A **const** is a promise that the value will not be modified once set.
const is a type qualifier.

A **static** variable is for the class, not per object. Memory is allocated only once per class, and every instance of that class uses it.
static is a storage specifier

8)

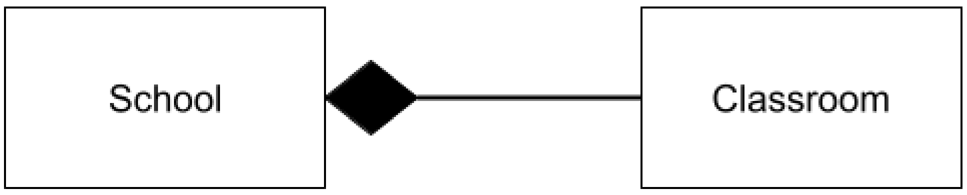
```
#include<iostream>

using namespace std;

template<typename T>
double Avg(T a, T b){
    return (a + b) / 2;
}

template<typename T>
double Avg(T *a, int size){
    double average;
    average = 0.0;

    for (int i = 0; i < size; i++){
        average += a[i];
    }

    average /= size;

    return average;
}

int main(){
    cout << Avg(20, 50)<<endl;
    double a[3] = {1.2, 3.5, 43.66};
    cout << Avg(a, 3)<<endl;
    int b[4] = {1, 3, 43, 14};
    cout << Avg(b, 4)<<endl;
}
```

9) **Aggregation** means one object *has* another object
For example, a classroom *has a* student



The classroom object has no breaking on the student object. The destruction of the classroom object has no effect on or destruction of the student object. Aggregation is represented by the diamond symbol.

Composition is a type of aggregation that describes ownership
For example, a School *contains* classrooms



If the school object is destroyed, the classroom object is also destroyed.. Composition is represented by the highlighted diamond symbol.