1.0 This lab demonstrates **memberwise assignment**

Note : *Memberwise assignment* is when the = operator may be used to assign one object's data to another object, or to initialize one object with another object's data. By default, each member of one object is copied to its counterpart in the other object.

```
#include <iostream>
#include "Rectangle.h"
using namespace std;

int main()
{
     // Define two Rectangle objects.
     Rectangle box1(10.0, 10.0);   // width = 10.0, length = 10.0
     Rectangle box2 (20.0, 20.0);  // width = 20.0, length = 20.0

     // Display each object's width and length.
     cout << "box1's width and length: " << box1.getWidth()
          << " " << box1.getLength() << endl;
     cout << "box2's width and length: " << box2.getWidth()
          << " " << box2.getLength() << endl << endl;

     // Assign the members of box1 to box2.
     box2 = box1;

     // Display each object's width and length again.
     cout << "box1's width and length: " << box1.getWidth()
          << " " << box1.getLength() << endl;
     cout << "box2's width and length: " << box2.getWidth()
          << " " << box2.getLength() << endl;

     return 0;
}
```

Activity: If you study this `main()` function, what do you think your `Rectangle.h` class should have? Discuss with your friends and create `Rectangle.h`

Remember the difference between assignment and initialization: assignment occurs between two objects that already exist, and initialization happens to an object being created.

2.0 **Copy constructor** : Most of the time, the default memberwise assignment behavior in C++ is perfectly acceptable. There are instances, however, where memberwise assignment cannot be used. Given the header file below:

```
#ifndef STUDENTTESTSCORES_H
#define STUDENTTESTSCORES_H
#include <string>
using namespace std;

const double DEFAULT_SCORE = 0.0;

class StudentTestScores
{
private:
     string studentName; // The student's name
     double *testScores; // Points to array of test scores
     int numTestScores;  // Number of test scores

     // Private member function to create an
     // array of test scores.
     void createTestScoresArray(int size)
     {
          }

public:
     // Constructor
     StudentTestScores(string name, int numScores)
     { studentName = name;
       createTestScoresArray(numScores); }

     // Destructor
     ~StudentTestScores()
     { delete [] testScores; }

     // The setTestScore function sets a specific
     // test score's value.
     void setTestScore(double score, int index)
     {  }

     // Set the student's name.
     void setStudentName(string name)
     {  }

     // Get the student's name.
     string getStudentName() const
     {  }
```

```
    // Get the number of test scores.
    int getNumTestScores() const
    { [                    ] }

    // Get a specific test score.
    double getTestScore(int index) const
    { [                    ] }
};
```

Activity: Discuss and fill in the blank space with appropriate codes to complete the header.

Note:

- The `studentName` attribute is a string object that holds a student's name. The `testScores` attribute is an `int` pointer. Its purpose is to point to a dynamically allocated `int` array that holds the student's test score. The `numTestScore` attribute is an `int` that holds the number of test scores.
- The `createTestScoresArray` private member function creates an array to hold the student's test scores. It accepts an argument for the number of test scores, assigns this value to the `numTestScores` attribute, and then dynamically allocates an `int` array for the `testScores` attribute. The `for` loop initializes each element of the array to the default value 0.0.
- A potential problem with this class lies in the fact that one of its members, `testScores` , is a pointer. The `createTestScoresArray` member function (called by the constructor) performs a critical operation with the pointer: it dynamically allocates a section of memory for the `testScores` array and assigns default values to each of its element. For instance, the following statement creates a `StudentTestScores` object named `student1` , whose `testScores` member references dynamically allocated memory holding an array of 5 double 's:

  ```
  StudentTestScores("Ajune Wanis Ismail", 5);
  ```

- The solution to this problem is to create a **copy constructor** for the object. A **copy constructor** is a special constructor that's called when an object is initialized with another object's data. It has the same form as other constructors, except it has a reference parameter of the same class type as the object itself.

Activity: Create a copy constructor for the `StudentTestScores` class. Write the `main` function to display the students name and scores. You should create two `StudentTestScores` objects, `student1` and `student2` with their 3 default test scores as shown here:

```
StudentTestScores student1 ("Ahmad Daniel", 3);
StudentTestScores student2 = student1;
```

3.0 **Operator Overloading -** redefine an existing operator's behavior

Although copy constructors solve the initialization problems inherent with objects containing pointer members, they do not work with simple assignment statements. Copy constructors are just that—constructors. They are only invoked when an object is created.

In order to change the way the assignment operator works, it must be overloaded. Operator overloading permits you to redefine an existing operator's behavior when used with a class object.

```
// Overloaded = operator
void operator=(const StudentTestScores &right)
{ delete [] testScores;
  studentName = right.studentName;
  numTestScores = right.numTestScores;
  testScores = new double[numTestScores];
  for (int i = 0; i < numTestScores; i++)
    testScores[i] = right.testScores[i]; }
```

In learning the mechanics of operator overloading, it is helpful to know that the following two statements do the same thing:

```
student2 = student1;          // Call operator= function
student2.operator=(student1); // Call operator= function
```

Activity: Amend the header by adding segment of code above to demonstrate the overloaded = operator. You do not need to amend the main function.

Overloading operator with return value

```
const StudentTestScores operator=(const StudentTestScores &right)
{ delete [] testScores;
  studentName = right.studentName;
  numTestScores = right.numTestScores;
  testScores = new double[numTestScores];
  for (int i = 0; i < numTestScores; i++)
      testScores[i] = right.testScores[i];
  return *this; }
```

This statement returns the value of a dereferenced pointer: `this` . But what is `this` ?
The `this` pointer is a special built-in pointer that is available to a class's member functions. It always points to the instance of the class making the function call.