ACTIVITY NO. 2

ARRAYS, POINTERS AND DYNAMIC MEMORY ALLOCATION	
Course Code: CPE010	**Program:** Computer Engineering
Course Title: Data Structures and Algorithms	**Date Performed:** 07/31/2025
Section: CPE21S4	**Date Submitted:** 08/07/2025
Name: Cabrera, Gabriel A.	**Instructor:** Engr. Jimlord Quejado

1. Objective(s)

- Demonstrate the use of dynamic memory allocation

2. Intended Learning Outcomes (ILOs)

After this module, the student should be able to:

a. Implement static and dynamic memory allocation
b. Create dynamically allocated objects using pointers and arrays
c. Solve programming problems using dynamic memory allocation, arrays and pointers

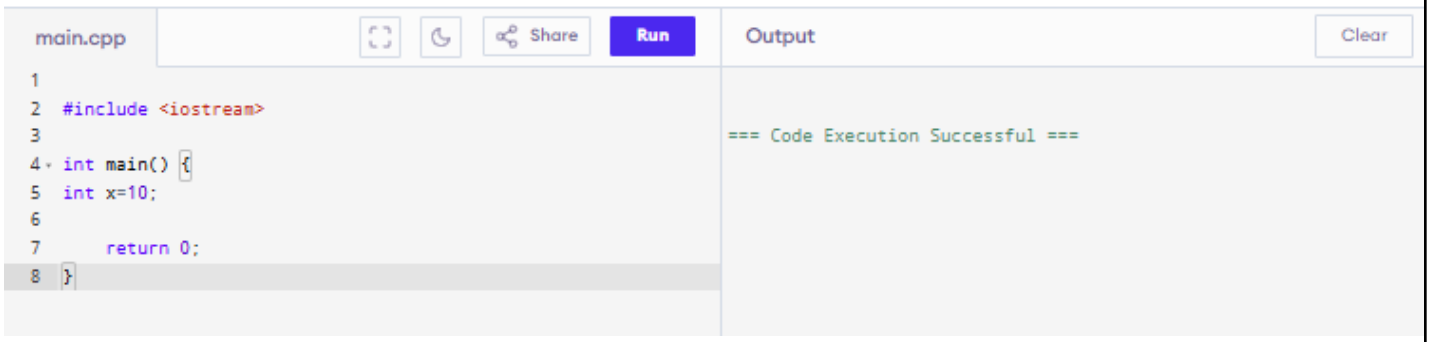
3. Discussion

Part A: Variables

Typical Variable Declaration

```
int x = 10;
```

Using the assignment operator, assign the value 10 to the memory address represented by x. During compilation, this variable name is translate to the memory address.

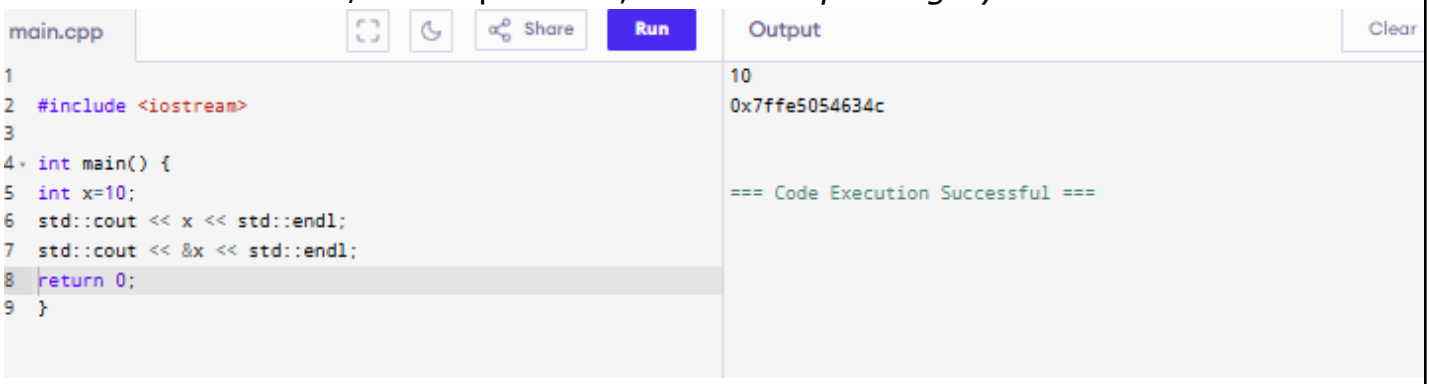


Reference Operator (&)

The reference operator (&) is used to retrieve memory addressed that the variable represents.

```
int x = 10;
std::cout << x << std::endl;
std::cout << &x << std::endl;
```

When used as a function/method parameter, this is called *passing by reference*.



Dereference Operator (*)

This operator (*) allows accessing a value at a particular memory location.

```
int x = 10;
std::cout << x << std::endl;
std::cout << &x << std::endl;
std::cout << *&x << std::endl;
```

main.cpp

Share Run

```
1
2  #include <iostream>
3
4  int main() {
5  int x=10;
6  std::cout << x << std::endl;
7  std::cout << &x << std::endl;
8  std::cout << *&x << std::endl;
9  return 0;
10 }
```

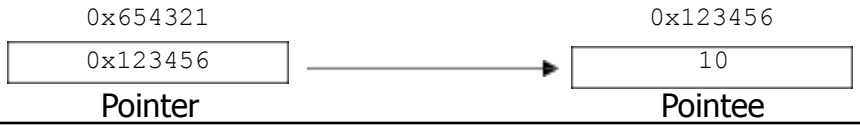
Output Clear

```
10
0x7ffdb0bb8bac
10


=== Code Execution Successful ===
```

Part B: Pointers

Simply put, a pointer is a variable that has the address of a memory location that contains data.



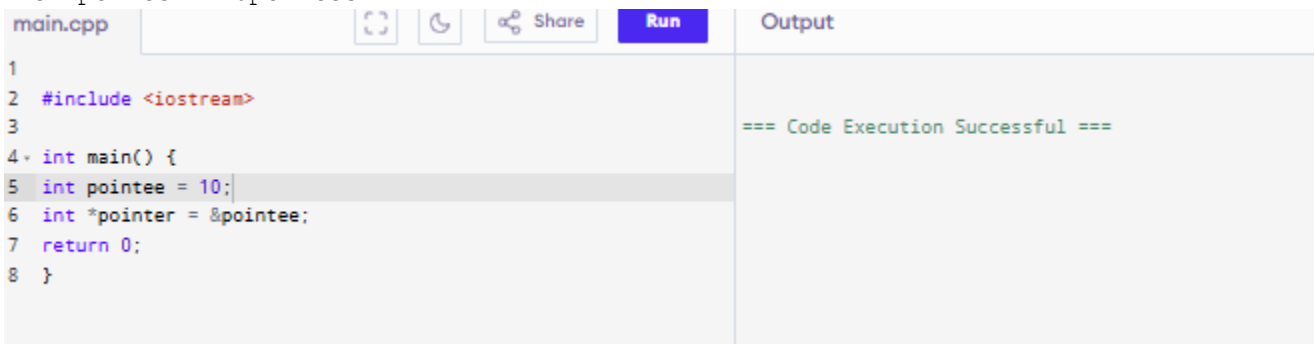
To declare the pointer:

- Indicate the variable type
- Put the dereference operator
- Put the name of the variable

```
type *var_name
int *intPtr
double *dbPtr
```

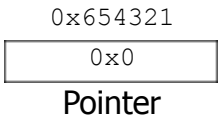
The diagram above has the accompanying code for implementation:

```
int pointee = 10
int *pointer = &pointee
```

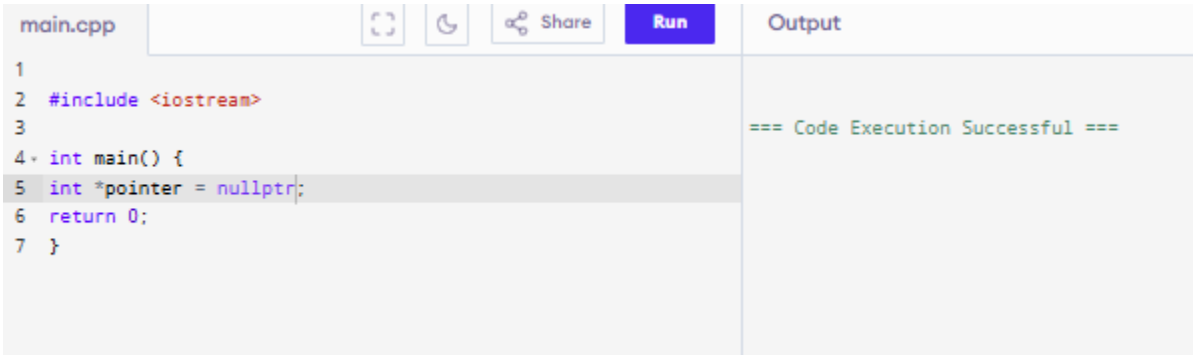


nullptr

When a pointer points to nothing, the keyword nullptr is used.




```
int *pointer = nullptr;
```



Part C: Dynamic Memory Allocation

So far, allocation has been done on the stack. When allocation is done during runtime, it uses the heap instead. This is dynamic memory allocation.

C++ Memory

4 Main Parts of Memory:

- Code
- Static (Global) Variables / Data
- Stack
- Heap

The heap and stack vary dynamically. The code and static are fixed in use. The code is also read only.

Memory

The new keyword

To perform dynamic memory allocation, we use the keyword for allocating memory called **new**. It allocates memory during runtime and allocated the memory for the given data in the heap.

```
pointerVariable = new type;
```

main.cpp | Share | Run | Output

```
1
2  #include <iostream>
3
4  int main() {
5      int *pointervariable=new int[5];
6  return 0;
7  }
```

```
=== Code Execution Successful ===
```

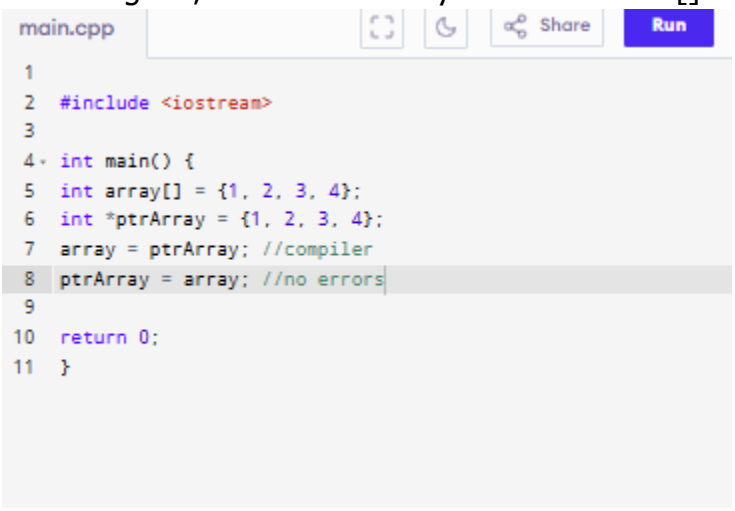
Arrays
In C++, the built-in arrays are created using

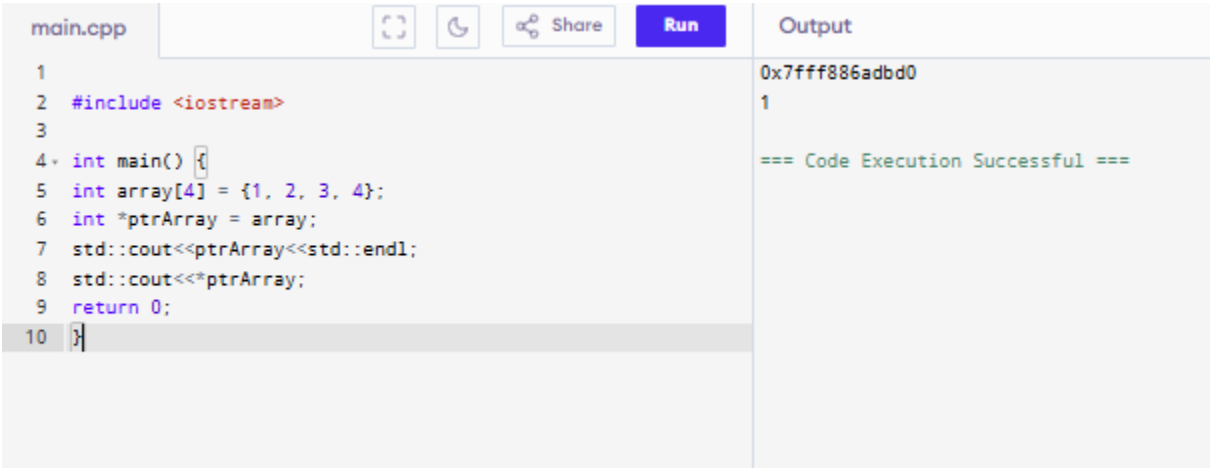
pointers! int array[] = {1, 2, 3, 4};

This code creates a pointer called array so that we can access the elements inside the array. The compiler knows that when the [] syntax is used, there will be an array of integer values. This syntax also makes it so that such arrays are declared const. Example:

```
int array[] = {1, 2, 3, 4};
int *ptrArray = {1, 2, 3, 4};
array = ptrArray; //compiler error
ptrArray = array; //no errors
```

Arrays declared with [] against those using pointers is that the ones declared using pointers can be reassigned, but not the arrays declared with [].





Dynamically Allocated Arrays
Arrays can be dynamically allocated:

```
int *array = new int[10];
```

main.cpp

```
1
2  #include <iostream>
3
4  int main() {
5  int *array=new int[10];
6  return 0;
7  }
```

Output

```
=== Code Execution Successful ===
```

Accessing these arrays on the heap is done the same way as before:

```
Array[index];
```

main.cpp

```
1
2  #include <iostream>
3
4  int main() {
5  int *array=new int[10];
6  std::cout<<array[5];
7  return 0;
8  }
```

Output

```
0

=== Code Execution Successful ===
```

But initialization is different:

```
int *array = {1, 2, 3, 4}; //error
```

main.cpp

```
1
2  #include <iostream>
3
4  int main() {
5  int *Array = {1, 2, 3, 4};
6
7  return 0;
8  }
```

Output

```
ERROR!
/tmp/W3Kbx8M1p8/main.cpp: In function 'int main()':
/tmp/W3Kbx8M1p8/main.cpp:5:6: error: scalar object 'Array'
    requires one element in initializer
    5 | int *Array = {1, 2, 3, 4};
      |      ^~~~~


=== Code Exited With Errors ===
```

The delete keyword

The **delete** operator does the opposite of the new operator. Instead of allocating, it deallocates assigned memory through the syntax:

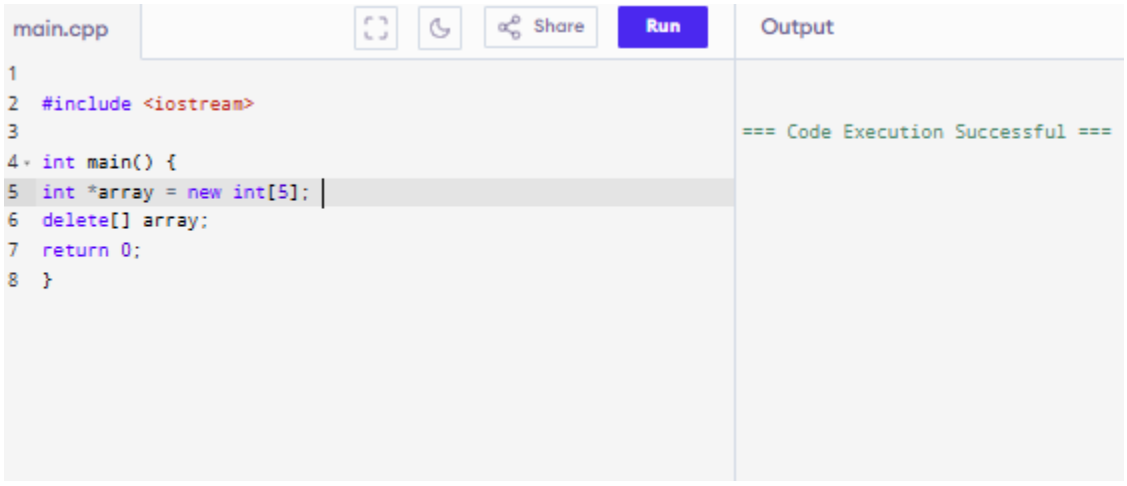
```
delete ptr;
```

General rule of thumb: for every **new** operator you must have a **delete** operation. This is to avoid a **memory leak**, which is caused by the failure to de-allocate memory that is pointed to by a pointer. This is caused by common mistakes such as:

- While in a loop, allocating memory but using delete on the pointer out of scope.
- Forgetting to delete data inside an object that is dynamically allocated.

How about for arrays?

```
int *array = new int[5];
delete[] array;
```



This delete operator is followed by a [] to indicate that we are deleting a block of memory.

Part D: Pointers with Objects/Classes

The use of pointers and dynamic allocation is normally used with the creation of

objects. Consider the declaration of the student class:

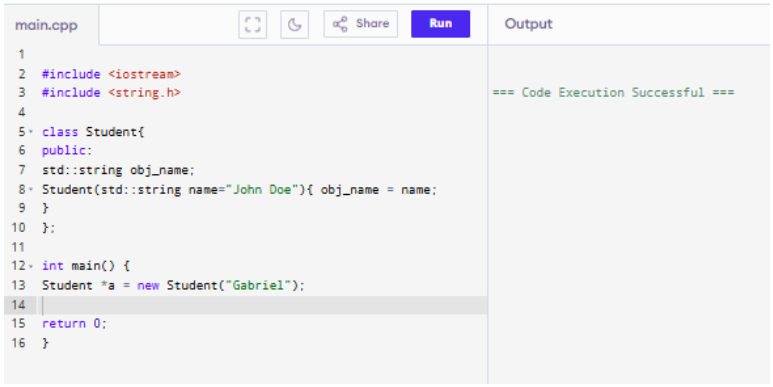
```
class Student{
    public:
    string obj_name;
    Student(string name="John Doe"){
        obj_name = name;
    }
};
```

We can dynamically create an object by:

```
Student *a = new Student;
```

This allocates a space for the Student object in the heap then calling the default constructor. Alternatively, you can pass an argument to the constructor, such that:

```
Student *a = new Student("Joshua");
```



Now, to access the data members of the class as shown in the code above. We can do the following:

```
(*a).data_member; //for data member
(*a).function(); //for functions
```

main.cpp

```
#include <iostream>
#include <string.h>

class Student{
public:
std::string obj_name;
Student(std::string name="John Doe"){ obj_name = name;
}
};

int main() {
Student *a = new Student("Gabriel");
std::cout<<(*a).obj_name; //for data member
return 0;
}
```

Output

```
Gabriel

=== Code Execution Successful ===
```

Alternatively, we can use the member access operator:

main.cpp

```
#include <iostream>
#include <string.h>

class Student{
public:
std::string obj_name;
Student(std::string name="John Doe"){ obj_name = name;
}
};

int main() {
Student *a = new Student("Gabriel");
std::cout<<a->obj_name; //for data member
return 0;
}
```

Output

```
Gabriel

=== Code Execution Successful ===
```

```
a->data_member;
a->function();
```

The Rule of Three

Some other important aspects about using pointers with classes and objects are the following known as the big three:

a. Destructors
b. Copy Constructor
c. Copy Assignment Operator

These also follow a good practice in programming that we call the rule of three. If you have to define one, define all of them.

Destructors
This member function is called that deletes an object. Some of the situations wherein we call destructors are when a function ends, when the program ends, when a block that contains local variables end, or when a delete operator is called. For every class, we have one destructor. Syntax is shown below:

```
Student::~Student(){
 /* Clean the data */
}
```

Copy Constructor
A copy constructor makes a copy of an existing instance. If we do not define the copy constructor, it is defined implicitly by the compiler per member of the source object.

It is important that we declare a copy constructor by passing in the class we want to copy as const.

```
Student::Student(const Student &copyStudent) {
 ...
}
```

The most common use for the copy constructor is when the class has raw pointers as member variables and we need to make a deep copy of the data.

Copy Assignment Operator
Often, we want to copy one object to another using the assignment operator. The program will implicitly create a copy assignment operator for you and do a member-wise copy. But again, we want to do a deep copy with an objects dynamically allocated data.

```
Student& Student::operator=(const Student& copy) {
 ...
}
```

4. Materials and Equipment

Personal Computer with C++ IDE
Recommended IDE:

- CLion (must use TIP email to download)
- DevC++ (use the embarcadero fork or configure to C++17)

5. Procedure

ILO A: Implement static and dynamic memory allocation & ILO B: Create dynamically allocated objects using pointers and arrays

In this activity, we will explore static and dynamic memory allocation through by utilizing arrays and other components mentioned beforehand, this will be for the implementation of a student list with the students' names and age. To begin, we will include pertinent libraries for input/output stream and strings (which we will be using in this activity).

```
#include <iostream>
#include <string.h>
```

Now we will create the student class with private attributes `studentName` and

```
studentAge. class Student{
private:
    std::string studentName;
    int studentAge;
```

Then, as discussed beforehand, we have to define the constructor and the big three: the destructor, copy constructor and copy assignment operator. We will assign default values for the parameters of our constructor in the event that it is uninitialized.

```
public:
    //constructor
    Student(std::string newName ="John Doe", int newAge=18){
        studentName = std::move(newName);
        studentAge = newAge;
        std::cout << "Constructor Called." << std::endl;
    };
```

The big three is then defined. We will add an output stream for each to observe when each of the following functions are implicitly or explicitly called.

```
    //deconstructor
    ~Student(){
        std::cout << "Destructor Called." << std::endl;
    }

    //Copy Constructor
    Student(const Student &copyStudent){
        std::cout << "Copy Constructor Called" << std::endl;
        studentName = copyStudent.studentName;
        studentAge = copyStudent.studentAge;
    }

    //Display Attributes
    void printDetails(){
        std::cout << this->studentName << " " << this->studentAge << std::endl;
    }

};
```

The driver program is now to be defined. We want to utilize the main function to simply show the static and dynamic allocation. However, we will explore the initial goal of creating instances of our Student class.

```
int main() {
    Student student1("Roman", 28);
    Student student2(student1);
    Student student3;
    student3 = student2;

    return 0;
}
```

Run the code and show the output. Include the screenshot in table 2-1 followed by your observation. Now, modify the driver program so that we have the array size of 5, an array of Student objects, the list of students' names and their age.

```
int main() {
    const size_t j = 5;

    Student studentList[j] = {};
    std::string namesList[j] = {"Carly", "Freddy", "Sam", "Zack", "Cody"};
    int ageList[j] = {15, 16, 18, 19, 16};
```

```
    return 0;
}
```

Run the code with the modified driver function and show the output. Include the screenshot in table 2-2 followed by your observation.

We have so far created static memory allocation through the use of the arrays. We will now dynamically allocate instances of the student class and store the newly created objects in the locations pointed to by our array.

```
int main() {
    const size_t j = 5;

    Student studentList[j] = {};
    std::string namesList[j] = {"Carly", "Freddy", "Sam", "Zack", "Cody"}; int
    ageList[j] = {15, 16, 18, 19, 16};

    for(int i = 0; i < j; i++){ //loop A
        Student *ptr = new Student(namesList[i], ageList[i]);
        studentList[i] = *ptr;
    }

    for(int i = 0; i < j; i++){ //loop B
        studentList[i].printDetails();
    }

    return 0;
}
```

Discuss what is done by loop A and loop B in table 2-3. Additionally, discuss the output and whether the functions are working as intended. If any corrections were made, further provide your modification and analysis in table 2-4.

6. Output

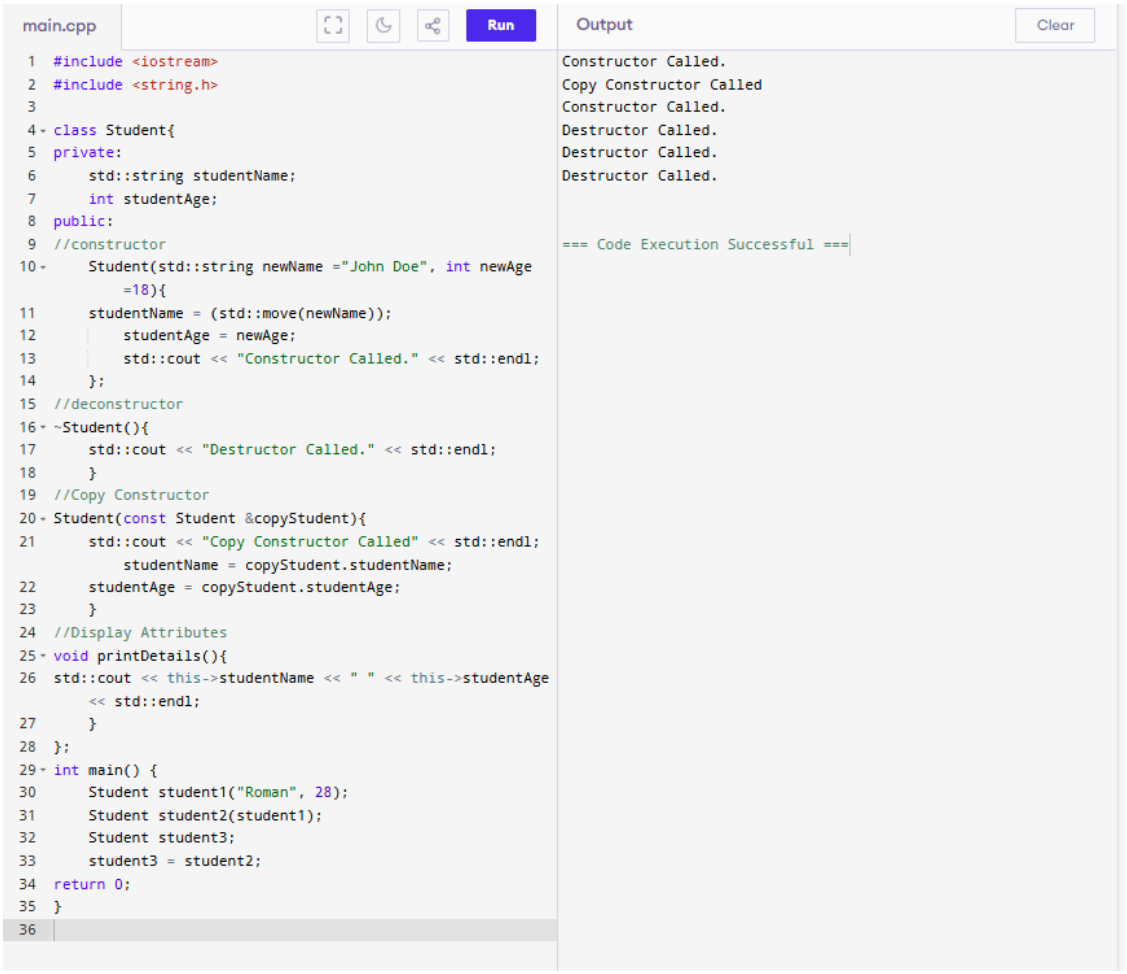
Screenshot	
Observation	The initial code tests if the given commands are working as intended, it first prints “Constructor Called” to show that the first student, Roman, was stored in the program, it prints “Copy Constructor Called” next because the next student (student2) referenced student1 to be stored in the program, the third student, even without any information, was also stored in the program and it printed another “Constructor Called” line. Finally, the program prints a “Destructor Called” line to make the user know that the pointers for the data given were deallocated.

Table 2-1. Initial Driver Program

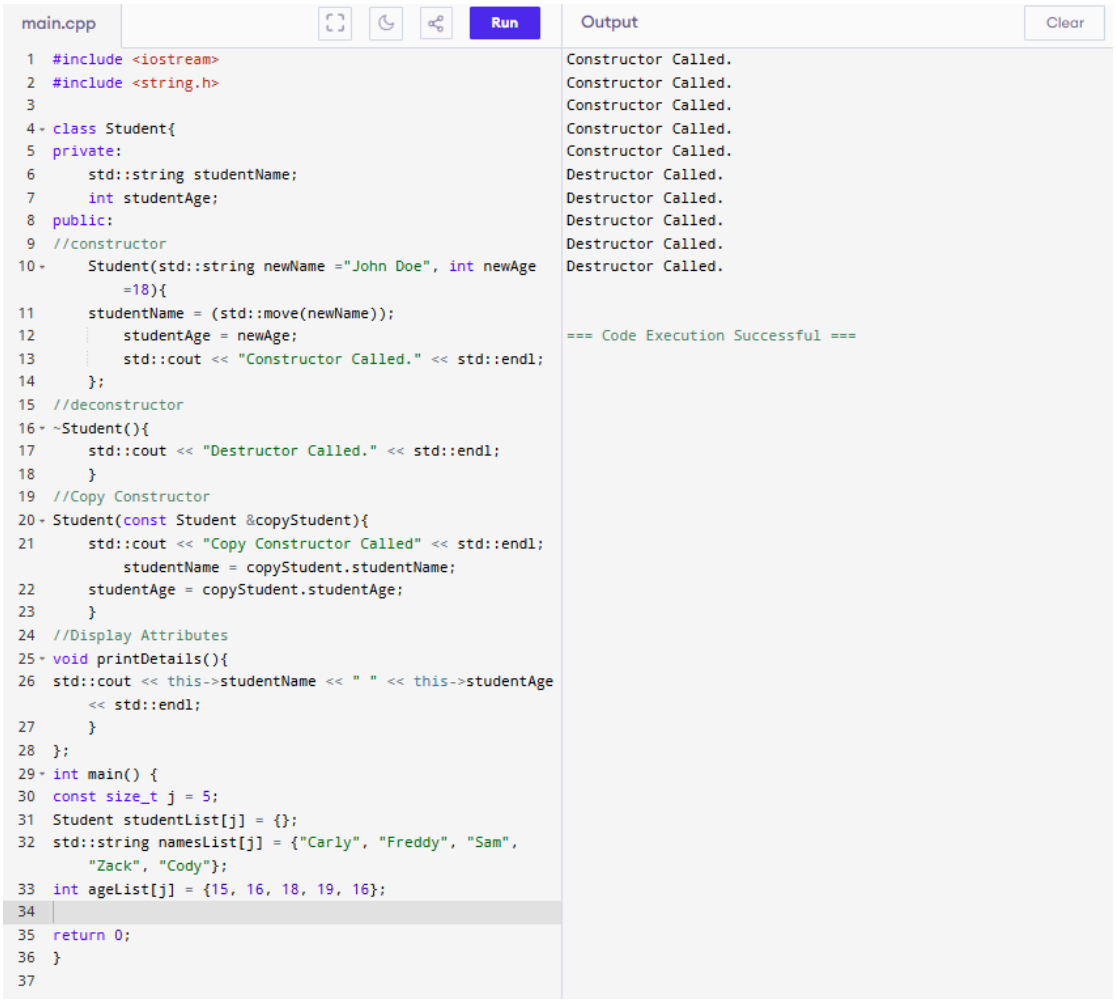
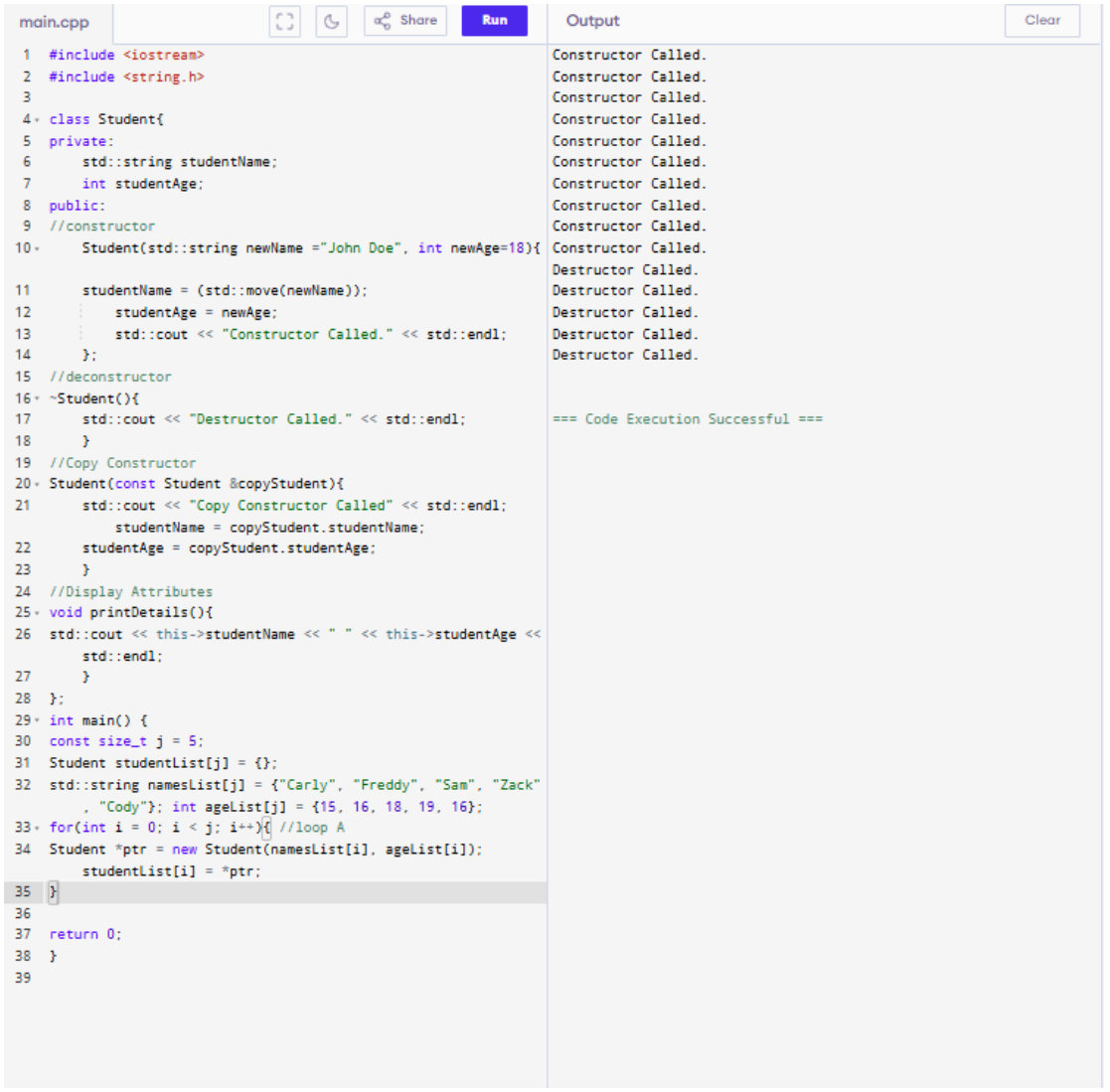
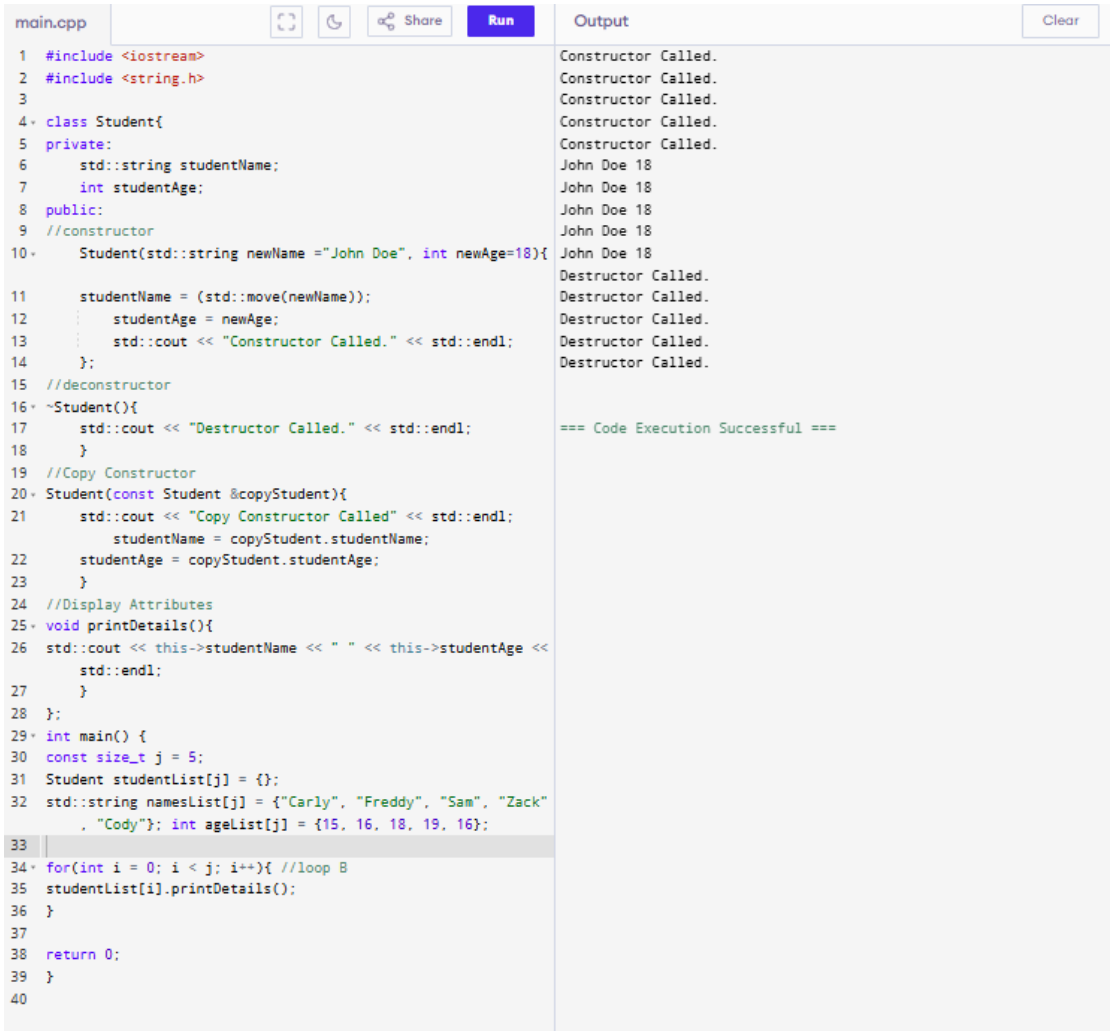
Screenshot	
Observation	Similar to the first driver program, this one also tests if the code works while looping through an array, it shows that it works by printing the “Constructor Called” and “Destructor Called” lines five times each to show that it allocated and deallocated five memory spaces for each student in the array.

Table 2-2. Modified Driver Program with Student Lists

Loop A	
Observation	The first loop of the program goes through both the namesList array and ageList array, this shows when the “Constructor Called” line was printed ten times. The “Destructor Called” line was only initiated five times, this is for every iteration in the loop which is declared by the constant variable j=5.

Loop B	
Observation	The “Constructor Called” lines were also printed for every iteration in the loop, the printDetails() function was also used in every iteration, but since there were no new students, which were supposed to be iterated by loop A, it uses the initially given newName which is John Doe from the constructor section of the code as well as his age which also came from the newAge variable.

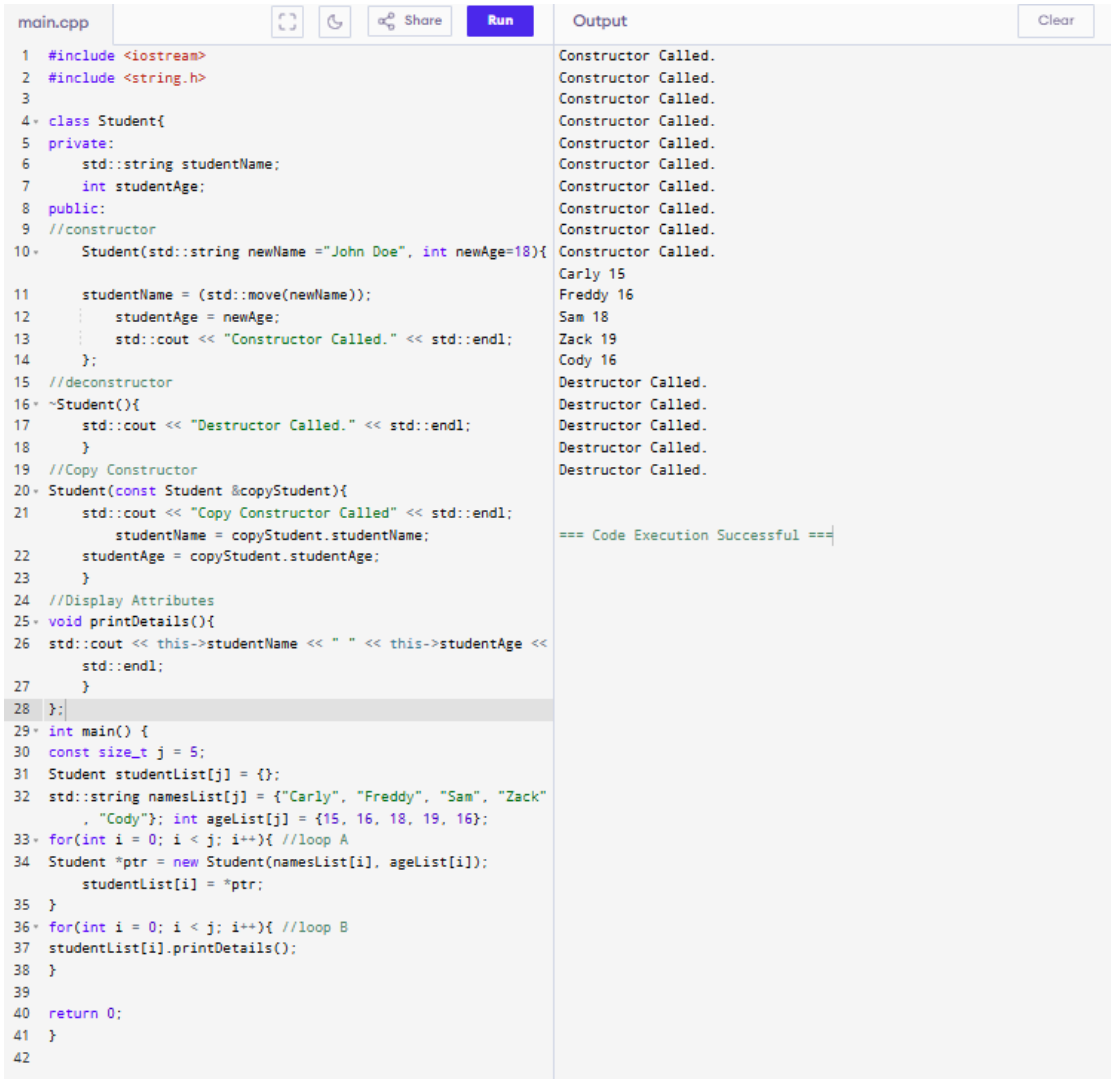
Output	
Observation	Combining the two loops, they can work together on iterating over every name and age in the given list while also printing them on every iteration. The first loop calls every name and age from namesList and ageList arrays and puts them on the studentList array while the second loop iterates and prints the newly updated studentsList array with the printDetails() function. The constructor and destructor lines were still called to show that they are working.

Table 2-3. Final Driver Program

Modifications	
Observation	As I observed in the current code, it doesn't need any modifications relevant to the activity, it successfully shows how to use pointers and memory allocation in various applications.

Table 2-4. Modifications/Corrections Necessary

7. Supplementary Activity

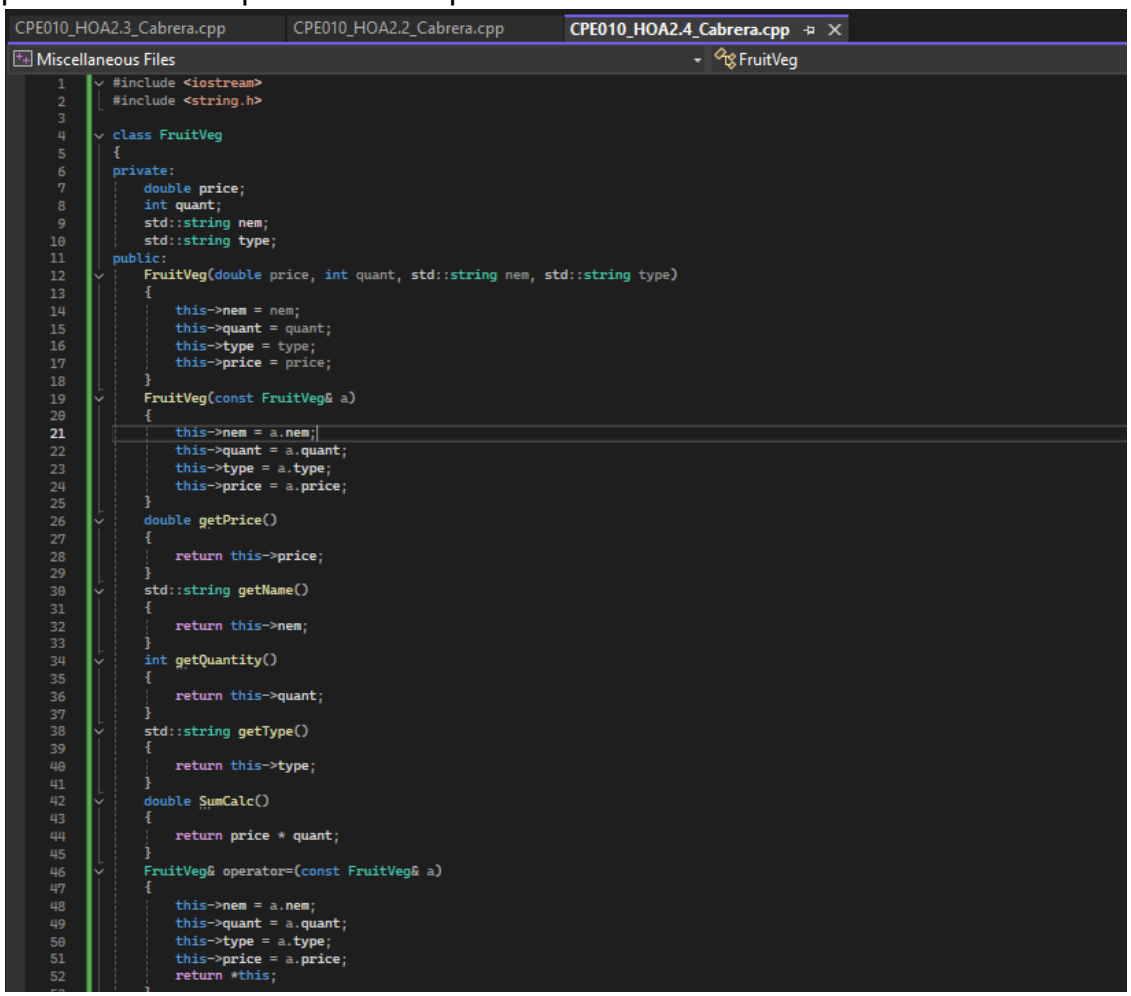
ILO C: Solve programming problems using dynamic memory allocation, arrays and

Jenna's Grocery List		
Apple	PHP 10	x7
Banana	PHP 10	x8
Broccoli	PHP 60	x12
Lettuce	PHP 50	x10

pointers

Jenna wants to buy the following fruits and vegetables for her daily consumption. However, she needs to distinguish between fruit and vegetable, as well as calculate the sum of prices that she has to pay in total.

Problem 1: Create a class for the fruit and the vegetable classes. Each class must have a constructor, deconstructor, copy constructor and copy assignment operator. They must also have all relevant attributes (such as name, price and quantity) and functions (such as calculate sum) as presented in the problem description above.




```cpp
#include <iostream>
#include <string.h>

class FruitVeg
{
private:
    double price;
    int quant;
    std::string nem;
    std::string type;
public:
    FruitVeg(double price, int quant, std::string nem, std::string type)
    {
        this->nem = nem;
        this->quant = quant;
        this->type = type;
        this->price = price;
    }
    FruitVeg(const FruitVeg& a)
    {
        this->nem = a.nem;
        this->quant = a.quant;
        this->type = a.type;
        this->price = a.price;
    }
    double getPrice()
    {
        return this->price;
    }
    std::string getName()
    {
        return this->nem;
    }
    int getQuantity()
    {
        return this->quant;
    }
    std::string getType()
    {
        return this->type;
    }
    double SumCalc()
    {
        return price * quant;
    }
    FruitVeg& operator=(const FruitVeg& a)
    {
        this->nem = a.nem;
        this->quant = a.quant;
        this->type = a.type;
        this->price = a.price;
        return *this;
```

I realized that since fruits and vegetables have the same attributes, I created only one class with the attributes for the name(nem), quantity(quant),price(price), and type(type). The type attribute is what I used to differentiate the fruits from vegetables in the list.

Problem 2: Create an array GroceryList in the driver code that will contain all items in Jenna's Grocery List. You must then access each saved instance and display all details about the items.

```
int main()
{
    int NoOfGroceryItems = 4;
    FruitVeg** GroceryList = new FruitVeg * [NoOfGroceryItems];

    GroceryList[0] = new FruitVeg(10.0, 7, "Apple", "Fruit  ");
    GroceryList[1] = new FruitVeg(10.0, 8, "Banana", "Fruit  ");
    GroceryList[2] = new FruitVeg(60.0, 12, "Broccoli", "Veggy  ");
    GroceryList[3] = new FruitVeg(50.0, 10, "Lettuce", "Veggy  ");
    std::cout << "List:\n\n";
    printFruitVeg(GroceryList, NoOfGroceryItems);
    std::cout << "\n\t\t\tTotal Sum " << "PHP" << sumtotal(GroceryList, NoOfGroceryItems);

    FruitVeg* Lettuce = GroceryList[3];
    for (int i = 3; i < NoOfGroceryItems - 1; i++)
    {
        GroceryList[i] = GroceryList[i + 1];
    }
    NoOfGroceryItems--;
    delete Lettuce;
    std::cout << "\nAfter removing Lettuce:\n\n";
    printFruitVeg(GroceryList, NoOfGroceryItems);

    return 0;
}
```

Manually adding each attribute of FruitVeg for each item in Jenna's grocery list.

Problem 3: Create a function TotalSum that will calculate the sum of all objects listed in Jenna's Grocery List.

```
    double SumCalc()
    {
        return price * quant;
    }
    FruitVeg& operator=(const FruitVeg& a)
    {
        this->nem = a.nem;
        this->quant = a.quant;
        this->type = a.type;
        this->price = a.price;
        return *this;
    }
    ~FruitVeg()
    {

    }
};
double sumtotal(FruitVeg** fruitveg, int sayz)
{
    double sum = 0.0;
    for (int i = 0; i < sayz; i++)
    {
        sum += fruitveg[i]->SumCalc();
    }
    return sum;
}
```

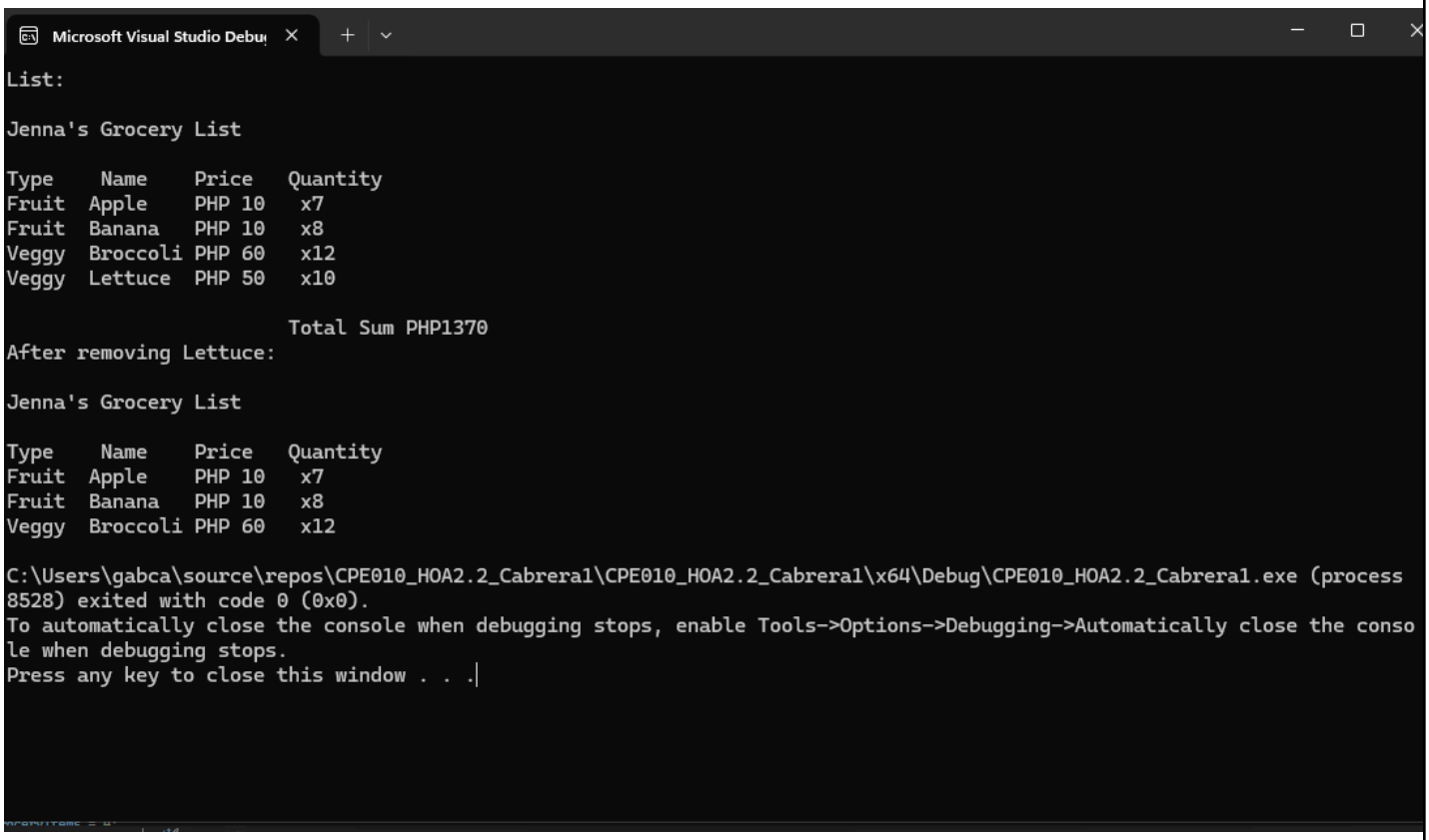
Basic function of getting the total of each item by multiplying their quantity value by their size value, then adding all of it by using iterations, looping for each item in the GroceryList array.

Problem 4: Delete the Lettuce from Jenna's GroceryList list and de-allocate the memory assigned.

```
    FruitVeg* Lettuce = GroceryList[3];
    for (int i = 3; i < NoOfGroceryItems - 1; i++)
    {
        GroceryList[i] = GroceryList[i + 1];
    }
    NoOfGroceryItems--;
    delete Lettuce;
    std::cout << "\nAfter removing Lettuce:\n\n";
    printFruitVeg(GroceryList, NoOfGroceryItems);

    return 0;
}
```

Referencing the fourth item(GroceryList[3]) in Jenna's list as Lettuce then deleting the said item and de-allocating the memory after.



Final output of the program with Jenna's full and modified list.

Complete Code:

CPE010_HOA2.4_Cabrera.cpp

Miscellaneous Files

```cpp
#include <iostream>
#include <string.h>

class FruitVeg
{
private:
    double price;
    int quant;
    std::string nem;
    std::string type;
public:
    FruitVeg(double price, int quant, std::string nem, std::string type)
    {
        this->nem = nem;
        this->quant = quant;
        this->type = type;
        this->price = price;
    }
    FruitVeg(const FruitVeg& a)
    {
        this->nem = a.nem;
        this->quant = a.quant;
        this->type = a.type;
        this->price = a.price;
    }
    double getPrice()
    {
        return this->price;
    }
    std::string getName()
    {
        return this->nem;
    }
    int getQuantity()
    {
        return this->quant;
    }
    std::string getType()
    {
        return this->type;
    }
    double SumCalc()
    {
        return price * quant;
    }
    FruitVeg& operator=(const FruitVeg& a)
    {
        this->nem = a.nem;
        this->quant = a.quant;
        this->type = a.type;
        this->price = a.price;
        return *this;
    }
    ~FruitVeg()
    {

    }
};
double sumtotal(FruitVeg** fruitveg, int sayz)
{
    double sum = 0.0;
    for (int i = 0; i < sayz; i++)
    {
        sum += fruitveg[i]->SumCalc();
    }
    return sum;
}
void printFruitVeg(FruitVeg** fruitveg, int sayz)
{
    std::cout << "Jenna's Grocery List\n\n";
    std::cout << "Type\t" << "Name\t" << "Price\t" << "Quantity\n";
    for (int i = 0; i < sayz; i++)
    {
        std::cout << fruitveg[i]->getType() << fruitveg[i]->getName() << "\tPHP "
            << fruitveg[i]->getPrice() << "\t x" << fruitveg[i]->getQuantity() << std::endl;
    }
}
int main()
{
    int NoOfGroceryItems = 4;
    FruitVeg** GroceryList = new FruitVeg * [NoOfGroceryItems];

    GroceryList[0] = new FruitVeg(10.0, 7, "Apple", "Fruit  ");
    GroceryList[1] = new FruitVeg(10.0, 8, "Banana", "Fruit  ");
    GroceryList[2] = new FruitVeg(60.0, 12, "Broccoli", "Veggy  ");
    GroceryList[3] = new FruitVeg(50.0, 10, "Lettuce", "Veggy  ");
    std::cout << "List:\n\n";
    printFruitVeg(GroceryList, NoOfGroceryItems);
    std::cout << "\n\t\t\tTotal Sum " << "PHP" << sumtotal(GroceryList, NoOfGroceryItems);

    FruitVeg* Lettuce = GroceryList[3];
    for (int i = 3; i < NoOfGroceryItems - 1; i++)
    {
        GroceryList[i] = GroceryList[i + 1];
    }
    NoOfGroceryItems--;
    delete Lettuce;
    std::cout << "\nAfter removing Lettuce:\n\n";
    printFruitVeg(GroceryList, NoOfGroceryItems);

    return 0;
}
```

75 % No issues found

8. Conclusion

With this activity, I was able to perform a wide range of tasks from the beginning like assigning basic pointers, arrays, and how to do dynamic and static memory allocation. Combining all the knowledge I learned from the discussion, I was also able to analyze and come up with observations for the tasks in the procedure, which teaches how important constructors, destructors, and copy constructors are, they are essential in making objects and implementing memory allocation for each created object. I was also able to make my own program by utilizing the things I learned from the discussion and procedures. The program can also be used as a template to other real world scenarios that use lists, like listing accounting balances and more grocery lists.

Overall, I think I did well in this activity, I'm sure that there are more ways to improve on this code and I'm excited to learn more on data structures and improve more.

9. Assessment Rubric