

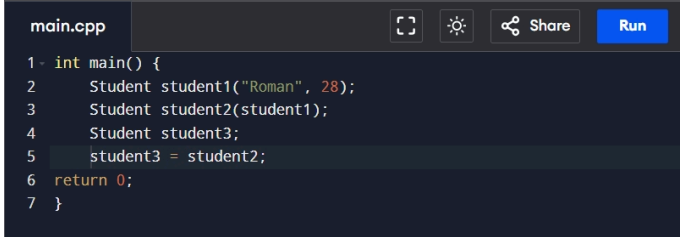

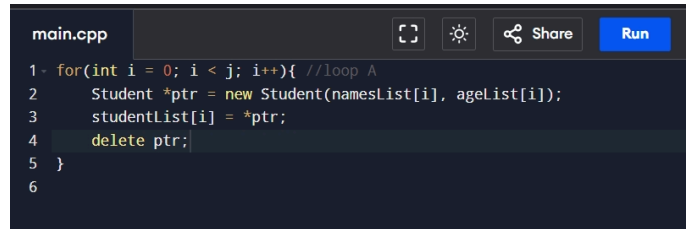
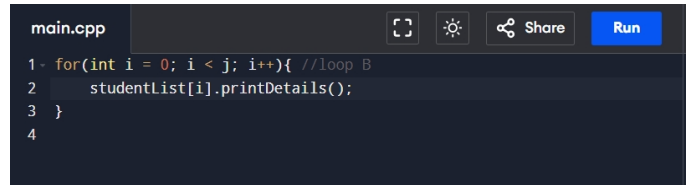

Activity No 2.1	
Hands-on Activity 2.1 Arrays, Pointers and Dynamic Memory Allocation	
Course Code: CPE010	Program: Computer Engineering
Course Title: Data Structures and Algorithms	Date Performed: 09/11/2024
Section: CPE21s4	Date Submitted: 09/11/2024
Name(s): Alexzander J. Reyes	Instructor: Prof. Maria Sayo
6. Output	
SCREENSHOT	 <pre> main.cpp 1 int main() { 2 Student student1("Roman", 28); 3 Student student2(student1); 4 Student student3; 5 student3 = student2; 6 return 0; 7 } </pre>
OBSERVATION	<p>The constructor is invoked when student1 is created. When student2 is initialized from student1, the copy constructor is used.</p> <p>The default constructor is called to create student3. The copy assignment operator assigns student2 to student3.</p> <p>Finally, the destructors for student1, student2, and student3 are automatically called as they go out of scope at the end of main().</p>
SCREENSHOT	 <pre> main.cpp 1 int main() { 2 const size_t j = 5; 3 Student studentList[j] = {}; 4 std::string namesList[j] = {"Carly", "Freddy", "Sam", "Zack", "Cody"}; 5 int ageList[j] = {15, 16, 18, 19, 16}; 6 return 0; 7 } </pre>
OBSERVATION	<p>When student1 is created, the constructor is called. The copy constructor is invoked when student2 is initialized using student1.</p> <p>For student3, the default constructor is used. Next, the copy assignment operator transfers the contents of student2 to student3.</p> <p>Finally, the destructors for student1, student2, and student3 are executed as they go out of scope at the end of the main() function.</p>

Table 2.1 Initial Driver Program

SCREENSHOT	
OBSERVATION	There is no dynamic memory allocation in this scenario; all objects are statically allocated. Each Student object in the studentList is created using the default constructor. Since no custom initialization occurs, there is no output from the constructors. If the printDetails function were called, it would display "John Doe 18" five times, which are the default values set for the Student class.

Table 2.2 Modified Driver Program with Student Lists

LOOP A	 <pre> main.cpp 1- for(int i = 0; i < j; i++){ //loop A 2 Student *ptr = new Student(namesList[i], ageList[i]); 3 studentList[i] = *ptr; 4 delete ptr; 5 } 6 </pre>
OBSERVATION	In Loop A, a new Student object is dynamically allocated for each entry in namesList and ageList. Each newly allocated object is then assigned to the corresponding index in the studentList array.
LOOP B	 <pre> main.cpp 1- for(int i = 0; i < j; i++){ //loop B 2 studentList[i].printDetails(); 3 } 4 </pre>
OBSERVATION	Loop B loops through the studentList array, invoking the printDetails() method for every Student object, which outputs each student's name and age.
OUTPUT	 <pre> Output /tmp/hhpNVACWnc.o Carly 15 Freddy 16 Sam 18 Zack 19 Cody 16 </pre>
OBSERVATION	This shows that the Student objects are successfully created, stored in the studentList array, and their details

are printed accurately.

Table 2.3 Final Driver Program

MODIFICATION

```
main.cpp
1- int main() {
2     const size_t j = 5;
3     Student studentList[j]; // Correct initialization for static
        allocation
4     string namesList[j] = {"Carly", "Freddy", "Sam", "Zack", "Cody"};
5     int ageList[j] = {15, 16, 18, 19, 16};
6
7     for (int i = 0; i < j; i++) {
8         Student* ptr = new Student(namesList[i], ageList[i]);
9         studentList[i] = *ptr;
10        delete ptr; // Free memory allocated by new
11    }
12
13    for (int i = 0; i < j; i++) {
14        studentList[i].printDetails(); // Added missing semicolon
15    }
16
17    return 0;
18 }
```

OBSERVATION

The code now properly deallocates memory for each dynamically allocated Student object, avoiding memory leaks. The delete ptr; statement in Loop A ensures that the memory allocated for each Student object created with new is released after being copied into the studentList array.

Table 2.4 Modifications/Corrections Necessary

7. Supplementary Activity

```
#include <iostream>
#include <vector>
#include <string>
#include <algorithm>
```

```
using namespace std;
```

```
// Base class for items
class Item {
protected:
    string name;
    double price;
    int quantity;
```

```
public:
    // Constructor
```

```

Item(const string& name, double price, int quantity)
    : name(name), price(price), quantity(quantity) {}

// Calculate total price
double calculateTotal() const {
    return price * quantity;
}

// Display item details
void display() const {
    cout << "Name: " << name << ", Price: PHP " << price
        << ", Quantity: " << quantity
        << ", Total: PHP " << calculateTotal() << endl;
}

// Check if name matches
bool isName(const string& nameToCompare) const {
    return name == nameToCompare;
}
};

// Fruit class inheriting from Item
class Fruit : public Item {
public:
    // Constructor
    Fruit(const string& name, double price, int quantity)
        : Item(name, price, quantity) {}
};

// Vegetable class inheriting from Item
class Vegetable : public Item {
public:
    // Constructor
    Vegetable(const string& name, double price, int quantity)
        : Item(name, price, quantity) {}
};

// Function to calculate the total sum of all items
double TotalSum(const vector<Item*>& list) {
    double sum = 0.0;
    for (const auto& item : list) {
        sum += item->calculateTotal();
    }
    return sum;
}

int main() {
    // Create grocery list
    vector<Item*> GroceryList = {
        new Fruit("Apple", 10.0, 7),
        new Fruit("Banana", 10.0, 8),

```

```

    new Vegetable("Broccoli", 60.0, 12),
    new Vegetable("Lettuce", 50.0, 10)
};

// Display all items
cout << "Initial Grocery List:\n";
for (const auto& item : GroceryList) {
    item->display();
}

// Calculate and display total sum
double totalSum = TotalSum(GroceryList);
cout << "Total Sum: PHP " << totalSum << endl;

// Remove Lettuce
auto it = remove_if(GroceryList.begin(), GroceryList.end(), [](Item* item) {
    bool toRemove = item->isName("Lettuce");
    if (toRemove) {
        delete item; // Clean up the memory
    }
    return toRemove;
});

GroceryList.erase(it, GroceryList.end());

// Display remaining items
cout << "\nAfter removing Lettuce:\n";
for (const auto& item : GroceryList) {
    item->display();
}

// Calculate and display new total sum
totalSum = TotalSum(GroceryList);
cout << "Total Sum after removal: PHP " << totalSum << endl;

// Cleanup remaining items
for (auto& item : GroceryList) {
    delete item;
}

return 0;
}

```

Output

```
/tmp/59eIIS7ySZ.o
```

```
Initial Grocery List:
```

```
Name: Apple, Price: PHP 10, Quantity: 7, Total: PHP 70
```

```
Name: Banana, Price: PHP 10, Quantity: 8, Total: PHP 80
```

```
Name: Broccoli, Price: PHP 60, Quantity: 12, Total: PHP 720
```

```
Name: Lettuce, Price: PHP 50, Quantity: 10, Total: PHP 500
```

```
Total Sum: PHP 1370
```

```
After removing Lettuce:
```

```
Name: Apple, Price: PHP 10, Quantity: 7, Total: PHP 70
```

```
Name: Banana, Price: PHP 10, Quantity: 8, Total: PHP 80
```

```
Name: Broccoli, Price: PHP 60, Quantity: 12, Total: PHP 720
```

```
Total Sum after removal: PHP 870
```

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

8. Conclusion

In this task, we investigated static and dynamic memory allocation by implementing classes for managing student and grocery lists. We discovered the importance of effective memory management techniques, including as constructors, destructors, copy constructors, and copy assignment operators. The exercise began with the establishment of a Student class, where we practiced initializing, copying, and assigning objects, and then progressed to dynamic memory allocation, illustrating how to handle object generation and deletion efficiently. We learned practical lessons about data handling in arrays by first building static arrays and then dynamically allocating memory for student objects. In the supplementary activity, we designed classes for fruits and vegetables to help students manage their grocery lists, including procedures for calculating total prices and handling item addition and removal. This section of the activity helped us grasp class design and memory management by distinguishing between different item types and calculating sums properly. Overall, the activity was effective in demonstrating essential programming ideas and enhancing our ability to manage memory and objects. Better handling of dynamic memory to prevent leaks, as well as the use of newer C++ features such as smart pointers, would increase code safety and efficiency. Future efforts could concentrate on improving resource management approaches and investigating additional C++ capabilities.

9. Assessment Rubric