Activity No. HOA 2.1	
ARRAYS, POINTERS AND DYNAMIC MEMORY ALLOCATION	
Course Code: CPE010	**Program:** Computer Engineering
Course Title: Data Structures and Algorithms	**Date Performed:** July 31,2025
Section: CPE21S4	**Date Submitted:** Aug 7,2025
Name(s): Bautista,Mariela S.	**Instructor: Engr.**Quejado

Discussion

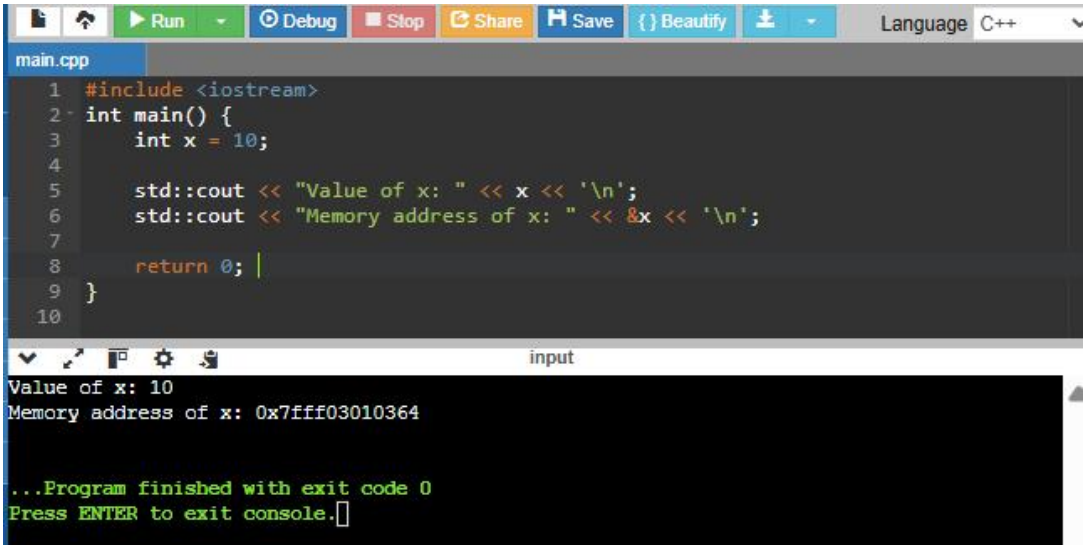
Part A: Variables

Reference Operator(&)

main.cpp

```cpp
#include<iostream>

int main()
{

  int x=10;
      std::cout << x << std::endl;
      std::cout << &x << std::endl;
    return 0;

  }
```

```
10
0x7fff12ab1bb4


...Program finished with exit code 0
Press ENTER to exit console.
```



Deference Operator (*)

main.cpp

```cpp
#include<iostream>

int main()
{

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

    return 0;

  }
```

```
10
0x7fffbcd26f54
10


...Program finished with exit code 0
Press ENTER to exit console.
```

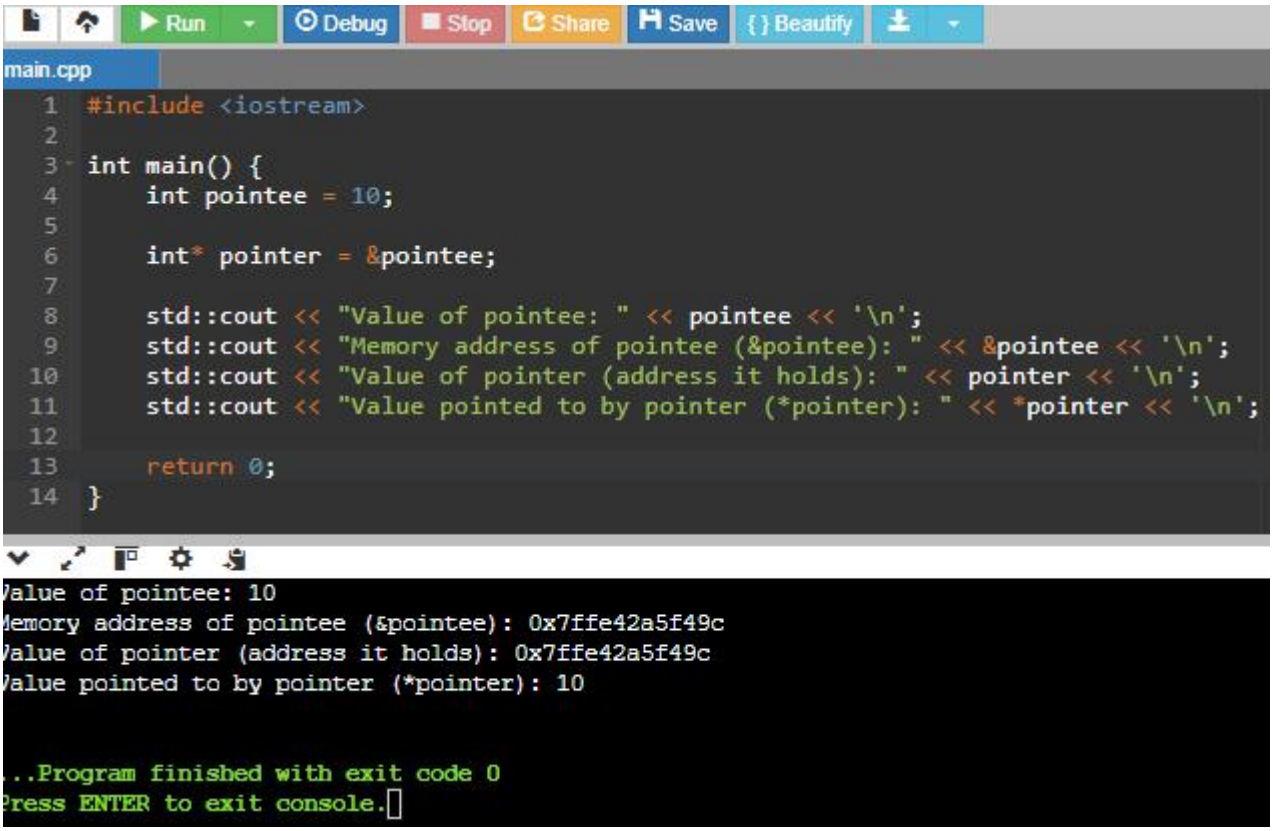
Run | Debug | Stop | Share | Save | {} Beautify

main.cpp

```cpp
#include <iostream>

int main() {
    int x = 10;


    std::cout << "Value of x: " << x << '\n';

    std::cout << "Memory address of x: " << &x << '\n';

    std::cout << "Value of x (using *&x): " << *&x << '\n';

    return 0;
}
```

```
Value of x: 10
Memory address of x: 0x7fff6bced494
Value of x (using *&x): 10


...Program finished with exit code 0
Press ENTER to exit console.
```

Part B: Pointers



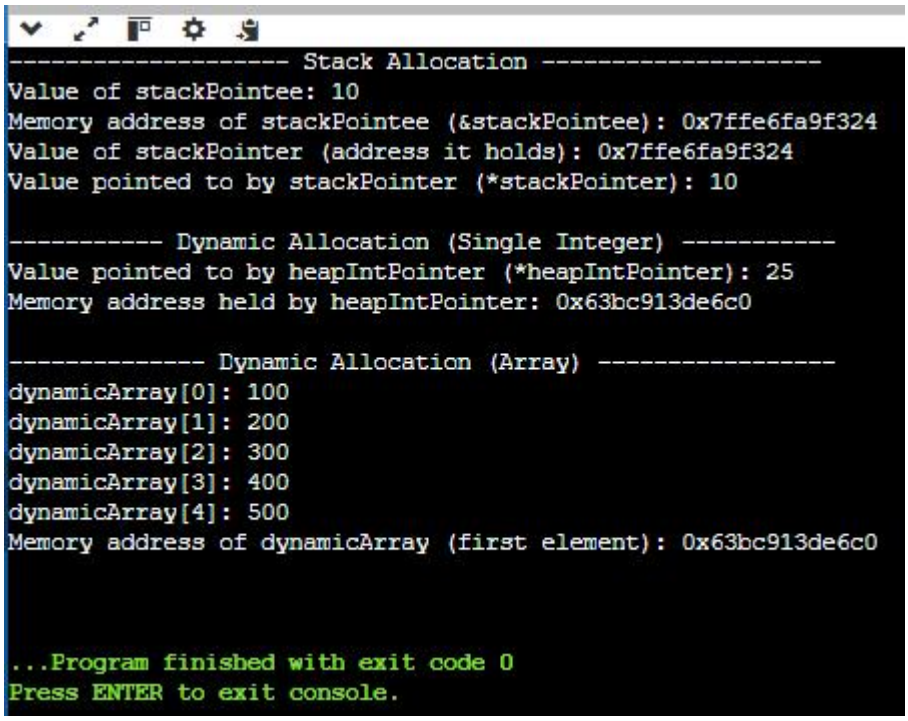
Part C: Dynamic Memory Allocation

Run | Debug | Stop | Share | Save | { } Beautify

main.cpp

```cpp
#include <iostream>

int main() {
    int stackPointee = 10;
    int* stackPointer = &stackPointee;

    std::cout << "-------------------- Stack Allocation --------------------" << '\n';
    std::cout << "Value of stackPointee: " << stackPointee << '\n';
    std::cout << "Memory address of stackPointee (&stackPointee): " << &stackPointee << '\n';
    std::cout << "Value of stackPointer (address it holds): " << stackPointer << '\n';
    std::cout << "Value pointed to by stackPointer (*stackPointer): " << *stackPointer << '\n';
    std::cout << '\n';

    int* heapIntPointer = new int;

    if (heapIntPointer == nullptr) {
        std::cerr << "Memory allocation failed for single integer!" << '\n';
        return 1;
    }

    *heapIntPointer = 25;

    std::cout << "----------- Dynamic Allocation (Single Integer) -----------" << '\n';
    std::cout << "Value pointed to by heapIntPointer (*heapIntPointer): " << *heapIntPointer << '\n';
    std::cout << "Memory address held by heapIntPointer: " << heapIntPointer << '\n';
    std::cout << '\n';


    delete heapIntPointer;
    heapIntPointer = nullptr;
    int arraySize = 5;
    int* dynamicArray = new int[arraySize];

    if (dynamicArray == nullptr) {
        std::cerr << "Memory allocation failed for array!" << '\n' ;
        return 1;
    }

    std::cout << "-------------- Dynamic Allocation (Array) ----------------" << '\n';
    for (int i = 0; i < arraySize; ++i) {
        dynamicArray[i] = (i + 1) * 100;
        std::cout << "dynamicArray[" << i << "]: " << dynamicArray[i] << '\n';
    }
    std::cout << "Memory address of dynamicArray (first element): " << dynamicArray << '\n';
    std::cout << '\n';

    delete[] dynamicArray;
    dynamicArray = nullptr;

    return 0;
}
```

Output:



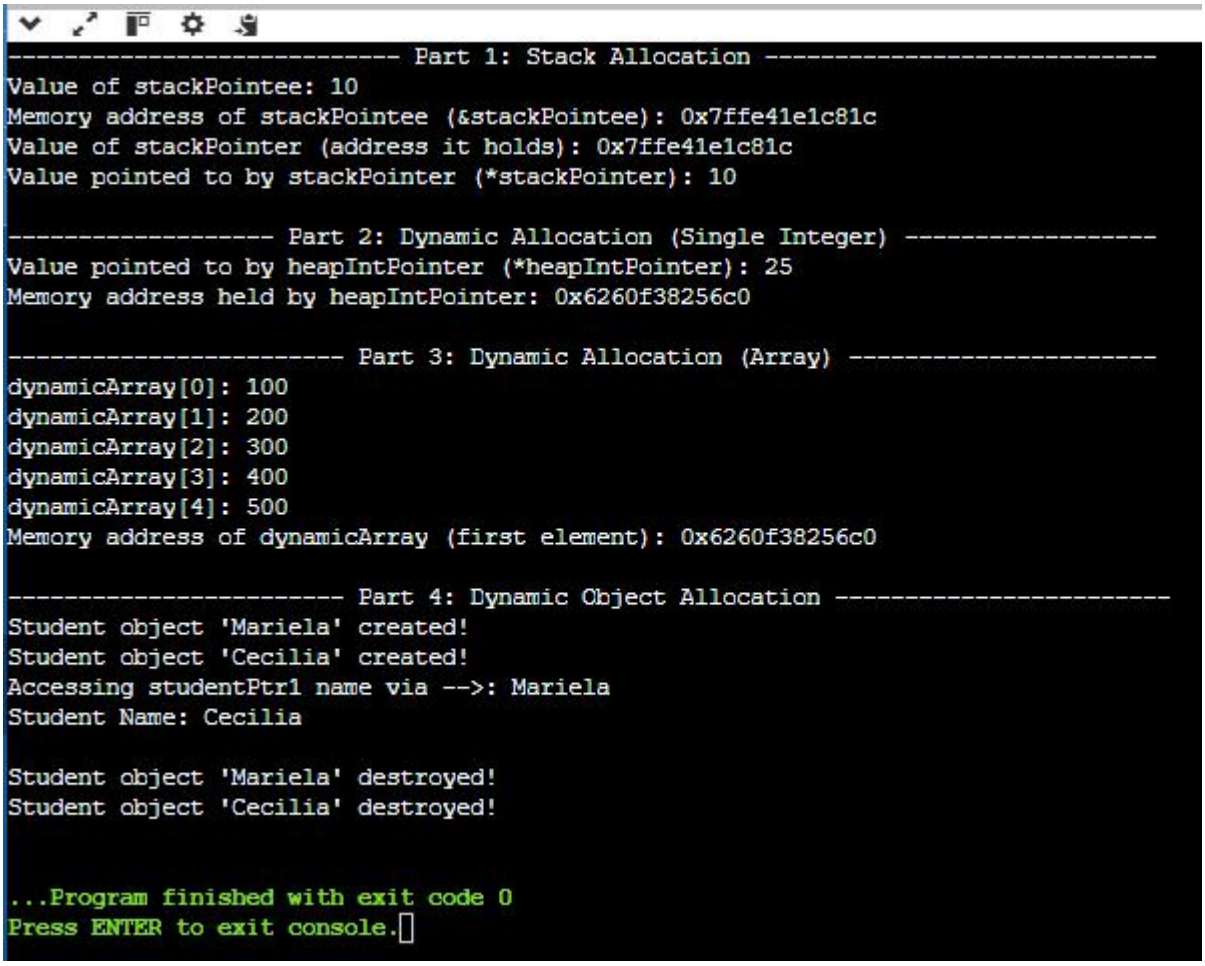
Part D: Pointers with Objects/Classes

Run | Debug | Stop | Share | Save | { } Beautify

main.cpp

```cpp
#include <iostream>
#include <string>
class Student {
public:
    std::string obj_name;

    Student(std::string name = "Johnny Bravo") {
        obj_name = name;
        std::cout << "Student object '" << obj_name << "' created!" << '\n';
    }

    ~Student() {
        std::cout << "Student object '" << obj_name << "' destroyed!" << '\n';
    }
    void displayInfo() {
        std::cout << "Student Name: " << obj_name << '\n';
    }
};

int main() {
    int stackPointee = 10;
    int* stackPointer = &stackPointee;

    std::cout << "---------------------------- Part 1: Stack Allocation ----------------------------" << '\n';
    std::cout << "Value of stackPointee: " << stackPointee << '\n';
    std::cout << "Memory address of stackPointee (&stackPointee): " << &stackPointee << '\n';
    std::cout << "Value of stackPointer (address it holds): " << stackPointer << '\n';
    std::cout << "Value pointed to by stackPointer (*stackPointer): " << *stackPointer << '\n';
    std::cout << '\n';

    int* heapIntPointer = new int;
    if (heapIntPointer == nullptr) {
        std::cerr << "Memory allocation failed for single integer!" << '\n';
        return 1;
    }
    *heapIntPointer = 25;

    std::cout << "------------------- Part 2: Dynamic Allocation (Single Integer) ------------------" << '\n';
    std::cout << "Value pointed to by heapIntPointer (*heapIntPointer): " << *heapIntPointer << '\n';
    std::cout << "Memory address held by heapIntPointer: " << heapIntPointer << '\n';
    std::cout << '\n';

    delete heapIntPointer;
    heapIntPointer = nullptr;

    int arraySize = 5;
    int* dynamicArray = new int[arraySize];

    if (dynamicArray == nullptr) {
        std::cerr << "Memory allocation failed for array!" << '\n';
        return 1;
    }

    std::cout << "----------------------- Part 3: Dynamic Allocation (Array) ----------------------" << '\n';
    for (int i = 0; i < arraySize; ++i) {
        dynamicArray[i] = (i + 1) * 100;
        std::cout << "dynamicArray[" << i << "]: " << dynamicArray[i] << '\n';
    }
    std::cout << "Memory address of dynamicArray (first element): " << dynamicArray << '\n';
    std::cout << '\n';

    delete[] dynamicArray;
    dynamicArray = nullptr;

    std::cout << "----------------------- Part 4: Dynamic Object Allocation ----------------------" << '\n';

    Student* studentPtr1 = new Student("Mariela");
    Student* studentPtr2 = new Student("Cecilia");

    std::cout << "Accessing studentPtr1 name via -->: " << studentPtr1->obj_name << '\n';
    studentPtr2->displayInfo();

    std::cout << '\n';

    delete studentPtr1;
    studentPtr1 = nullptr;

    delete studentPtr2;
    studentPtr2 = nullptr;

    return 0;
}
```

Output:


---------- Part 1: Stack Allocation ----------
Value of stackPointee: 10
Memory address of stackPointee (&stackPointee): 0x7ffe41e1c81c
Value of stackPointer (address it holds): 0x7ffe41e1c81c
Value pointed to by stackPointer (*stackPointer): 10
---------- Part 2: Dynamic Allocation (Single Integer) ----------
Value pointed to by heapIntPointer (*heapIntPointer): 25
Memory address held by heapIntPointer: 0x6260f38256c0
---------- Part 3: Dynamic Allocation (Array) ----------
dynamicArray[0]: 100
dynamicArray[1]: 200
dynamicArray[2]: 300
dynamicArray[3]: 400
dynamicArray[4]: 500
Memory address of dynamicArray (first element): 0x6260f38256c0
---------- Part 4: Dynamic Object Allocation ----------
Student object 'Mariela' created!
Student object 'Cecilia' created!
Accessing studentPtr1 name via -->: Mariela
Student Name: Cecilia
Student object 'Mariela' destroyed!
Student object 'Cecilia' destroyed!
...Program finished with exit code 0
Press ENTER to exit console.

Output: Table 2-1. Initial Driver Program

Screenshot

Run | Debug | Stop | Share | Save | { } Beautify

main.cpp

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

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

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

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

    Student& operator=(const Student& copy) {
        std::cout << "Copy Assignment Operator Called for " << copy.studentName << "." << std::endl;
        if (this != &copy) { // Self-assignment check
            studentName = copy.studentName;
            studentAge = copy.studentAge;
        }
        return *this;
    }

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

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

    std::cout << "\nDetails after initial driver program:" << std::endl;
    student1.printDetails();
    student2.printDetails();
    student3.printDetails();

    return 0;
}
```

```
Constructor Called for Roman.
Copy Constructor Called for Roman.
Constructor Called for John Doe.
Copy Assignment Operator Called for Roman.

Details after initial driver program:
Roman 28
Roman 28
Roman 28
Destructor Called for Roman.
Destructor Called for Roman.
Destructor Called for Roman.


...Program finished with exit code 0
Press ENTER to exit console.
```

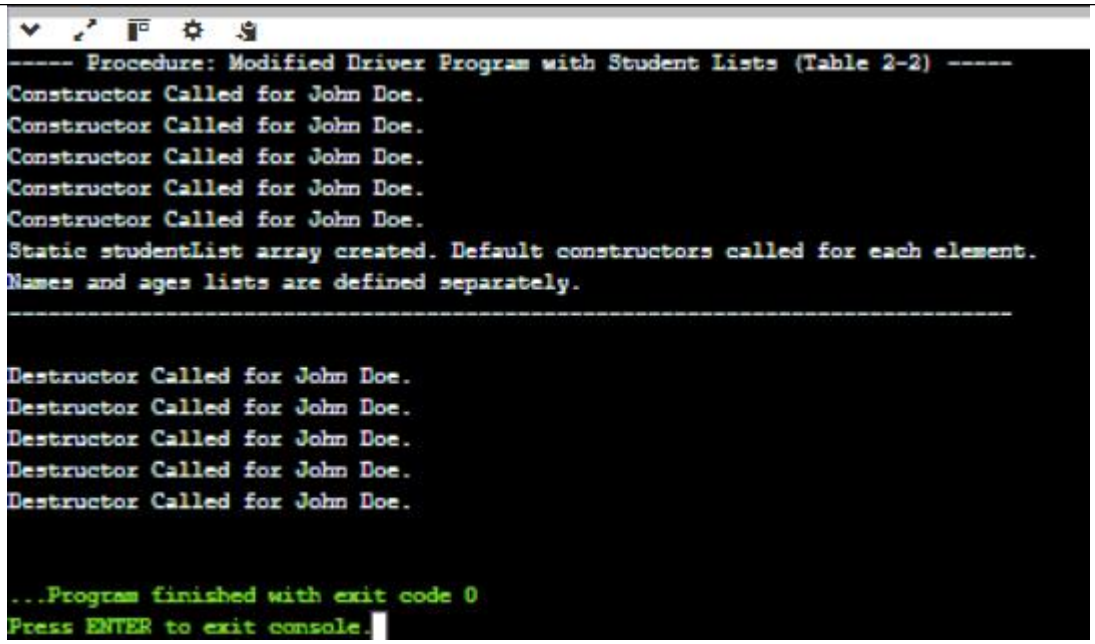
Observation	This part of the code shows how Student objects are created and copied, whether it happens directly or behind the scenes. When student1 is created, it uses the constructor with specific values. Then student2(student1) calls the copy constructor, which basically makes a clone of student1. Meanwhile, student3 is created using the default constructor. When we assign student2 to student3 using student3 = student2, the copy assignment operator takes care of copying the data. If you look at the output from printDetails(), all three objects — student1, student2, and student3 — end up with the same data: "Roman 28". After main() finishes, the destructors are called in reverse order, starting from student3, showing how memory is automatically cleaned up for stack-allocated objects.

Output: Table 2-2.Modified Driver Program with Student Lists

Screenshot

main.cpp

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

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

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

    ~Student(){
        std::cout << "Destructor Called for " << studentName << "." << std::endl;
    }

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

    Student& operator=(const Student& copy) {
        std::cout << "Copy Assignment Operator Called for " << copy.studentName << "." << std::endl;
        if (this != &copy) { // Self-assignment check
            studentName = copy.studentName;
            studentAge = copy.studentAge;
        }
        return *this;
    }

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

int main() {
    std::cout << "----- Procedure: Modified Driver Program with Student Lists (Table 2-2) -----" << std::endl;
    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};

    std::cout << "Static studentList array created. Default constructors called for each element." << std::endl;
    std::cout << "Names and ages lists are defined separately." << std::endl;
    std::cout << "--------------------------------------------------------------------------------" << std::endl << std::endl;

    return 0;
}
```

	
Observations	In this section, we're looking at how a static array of Student objects is set up. When Student studentList[j] = {}; runs, the program automatically calls the default constructor for each of the five Student objects in that array. So you'll see five "Constructor Called for John Doe" messages. The namesList and ageList are also created here, but they are just separate lists of names and ages. They don't actually change the Student objects in studentList at this point. When the main function finishes, the destructors are called for all the Student objects in the array, which shows how memory is handled for stack-allocated objects.

Output 2-3. Final Driver Program

Loop A	main.cpp ```for(int i = 0; i < j; i++){ //Loop A Student *ptr = new Student(namesList[i], ageList[i]); studentList[i] = *ptr; }```
Observation	Each iteration creates a Student object on the heap using values from the lists, then copies its contents into the stack-based array, triggering the copy assignment operator. Deleting the heap object right after shows intentional memory management, highlighting how temporary dynamic objects can still play a role in stack-based assignments.
Loop B	main.cpp ```for(int i = 0; i < j; i++){ //Loop B studentList[i].printDetails(); }```
Observation	This loop is pretty straightforward. It just goes through each Student object that's now stored in your studentList array (on the stack, holding the copied data from Loop A) and calls its printDetails() function. It will print the name and age of each student.

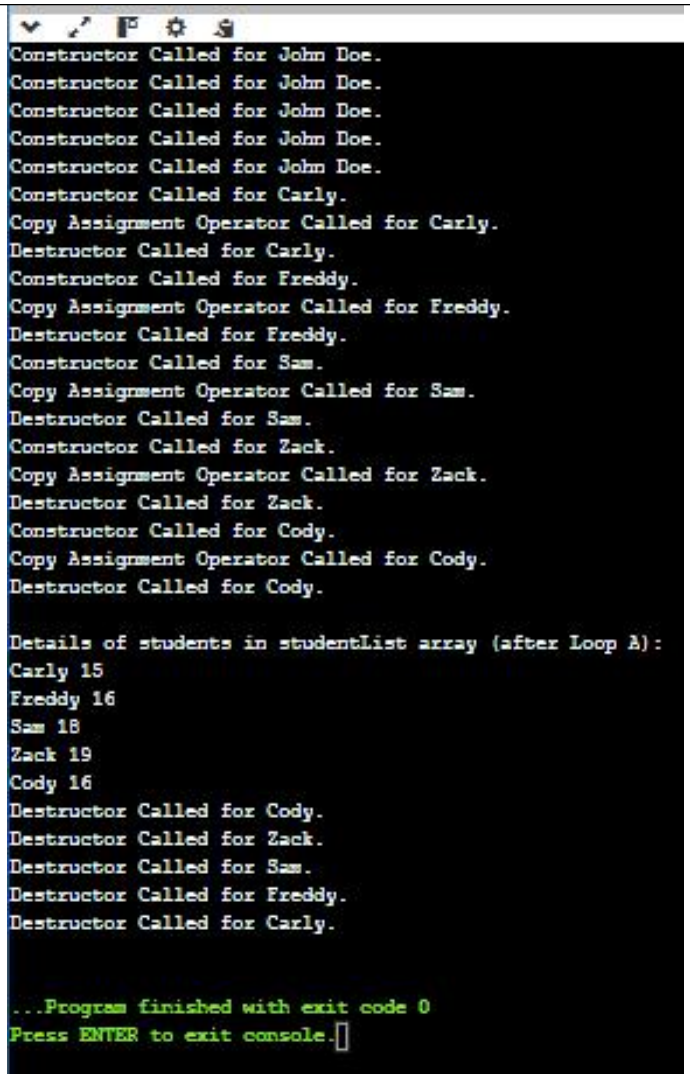
Output	main.cpp ```cpp #include <iostream> #include <string> class Student { private: std::string studentName; int studentAge; public: Student(std::string newName = "John Doe", int newAge = 18) : studentName(std::move(newName)), studentAge(newAge) { std::cout << "Constructor Called for " << studentName << "." << std::endl; } ~Student() { std::cout << "Destructor Called for " << studentName << "." << std::endl; } Student(const Student& copyStudent) : studentName(copyStudent.studentName), studentAge(copyStudent.studentAge) { std::cout << "Copy Constructor Called for " << studentName << "." << std::endl; } Student& operator=(const Student& copy) { std::cout << "Copy Assignment Operator Called for " << copy.studentName << "." << std::endl; if (this != &copy) { studentName = copy.studentName; studentAge = copy.studentAge; } return *this; } void printDetails() { std::cout << this->studentName << " " << this->studentAge << std::endl; } }; 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++){ Student *ptr = new Student(namesList[i], ageList[i]); studentList[i] = *ptr; / delete ptr; ptr = nullptr; } std::cout << "\nDetails of students in studentList array (after Loop A):" << std::endl; for(int i = 0; i < j; i++){ studentList[i].printDetails(); } return 0; } ```
Observation	Loop A creates temporary heap-based Student objects that get copied into a stack-allocated array, shown by the sequence of constructor, copy assignment, and destructor calls for each student. Loop B confirms the copied data is intact by printing each student's details, but overall, the logic is inefficient since dynamic allocation is used without keeping the actual dynamic objects.

Output 2-4.Modifications/Corrections Necessary

Modifications	Instead of creating temporary heap-allocated objects just to copy them into the array, directly initialize the studentList elements using the constructor. This skips the whole allocate-copy-delete cycle.

main.cpp

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

class Student {
private:
    std::string name;
    int age;

public:
    Student() : name("John Doe"), age(0) {
        std::cout << "Constructor Called for " << name << ".\n";
    }

    Student(const std::string& name, int age) : name(name), age(age) {
        std::cout << "Constructor Called for " << this->name << ".\n";
    }

    Student(const Student& other) : name(other.name), age(other.age) {
        std::cout << "Copy Constructor Called for " << name << ".\n";
    }

    Student& operator=(const Student& other) {
        if (this != &other) {
            name = other.name;
            age = other.age;
            std::cout << "Copy Assignment Operator Called for " << name << ".\n";
        }
        return *this;
    }

    ~Student() {
        std::cout << "Destructor Called for " << name << ".\n";
    }

    void printDetails() const {
        std::cout << name << " " << age << "\n";
    }
};

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};

    // Loop A: allocate and copy
    for (int i = 0; i < j; i++) {
        Student* ptr = new Student(namesList[i], ageList[i]);
        studentList[i] = *ptr;
        delete ptr;
    }

    // Loop B: print
    std::cout << "\nDetails of students in studentList array (after Loop A):\n";
    for (int i = 0; i < j; i++) {
        studentList[i].printDetails();
    }

    return 0;
}
```

	
Observations	By changing it to studentList_dynamic_pointers, I'm now actually storing the memory addresses of each Student created on the heap. It finally makes sense why I need to use -> — I'm not working with objects directly anymore, but with their locations in memory. What really matters here is the cleanup loop I added; for every new, there's a matching delete, so I'm not leaving anything behind in memory. This feels more intentional and aligned with how dynamic memory allocation should be done in C++.

7. Supplementary Activity

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

Jenna's Grocery List

Apple PHP 10 x7

Banana PHP 10 x8

Broccoli PHP 60 x12

Lettuce PHP 50 x10

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.

** I created two classes, Fruit and Vegetable, each with the proper constructors, destructors, copy constructors, and copy assignment operators. Attributes like name, price, and quantity are set and used in a calculateSum() method.

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.

** In the main code, I declared a dynamic array GroceryList to store both fruits and vegetables using pointers for dynamic memory allocation. I accessed each element using the arrow operator (->) and printed their details.

Problems 1 n 2;

```
int main() {
    const int MAX_ITEMS = 10;
    GroceryItem* GroceryList[MAX_ITEMS];
    int itemCount = 0;

    GroceryList[itemCount++] = new Fruit("Apple", 10, 7);
    GroceryList[itemCount++] = new Fruit("Banana", 10, 8);
    GroceryList[itemCount++] = new Vegetable("Broccoli", 60, 12);
    GroceryList[itemCount++] = new Vegetable("Lettuce", 50, 10);

    cout << "\n--- Grocery List ---\n";
    for (int i = 0; i < itemCount; ++i) {
        GroceryList[i]->printDetails();
    }
```

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

**The TotalSum() function loops through all items in GroceryList, calling calculateSum() on each one. The total price before deleting anything was correctly computed.

```
    cout << "\nTotal before deletion: ₱" << TotalSum(GroceryList, itemCount) << endl;
```

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

To delete Lettuce, I traversed the list, deallocated its memory using delete, and then removed its pointer from the array, ensuring proper cleanup.

```
    DeleteItem(GroceryList, itemCount, "Lettuce");

    cout << "\n--- Updated Grocery List ---\n";
    for (int i = 0; i < itemCount; ++i) {
        GroceryList[i]->printDetails();
    }

    cout << "\nTotal after deletion: ₱" << TotalSum(GroceryList, itemCount) << endl;


    for (int i = 0; i < itemCount; ++i) {
        delete GroceryList[i];
    }

    return 0;
}
```

Here is the complete code;

main.cpp

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

// Base class
class GroceryItem {
protected:
    string name;
    double price;
    int quantity;
public:
    GroceryItem(string n = "", double p = 0, int q = 0)
        : name(n), price(p), quantity(q) {
        cout << "[Constructor] " << name << endl;
    }

    virtual ~GroceryItem() {
        cout << "[Destructor] " << name << endl;
    }

    GroceryItem(const GroceryItem& other)
        : name(other.name), price(other.price), quantity(other.quantity) {
        cout << "[Copy Constructor] " << name << endl;
    }

    GroceryItem& operator=(const GroceryItem& other) {
        if (this != &other) {
            name = other.name;
            price = other.price;
            quantity = other.quantity;
            cout << "[Copy Assignment] " << name << endl;
        }
        return *this;
    }

    virtual double calculateSum() const {
        return price * quantity;
    }

    virtual void printDetails() const {
        cout << name << " - ₱" << price << " x" << quantity
             << " = ₱" << calculateSum() << endl;
    }

    string getName() const {
        return name;
    }
};

// Derived class: Fruit
class Fruit : public GroceryItem {
public:
    Fruit(string n = "", double p = 0, int q = 0)
        : GroceryItem(n, p, q) {}

    ~Fruit() {
        cout << "[Fruit Destructor] " << name << endl;
    }
};

// Derived class: Vegetable
class Vegetable : public GroceryItem {
public:
    Vegetable(string n = "", double p = 0, int q = 0)
        : GroceryItem(n, p, q) {}

    ~Vegetable() {
        cout << "[Vegetable Destructor] " << name << endl;
    }
};

double TotalSum(GroceryItem* list[], int size) {
    double total = 0;
    for (int i = 0; i < size; ++i) {
        if (list[i] != nullptr) {
```

```
            total += list[i]->calculateSum();
        }
    }
    return total;
}

void DeleteItem(GroceryItem* list[], int& size, const string& target) {
    for (int i = 0; i < size; ++i) {
        if (list[i] != nullptr && list[i]->getName() == target) {
            delete list[i];
            for (int j = i; j < size - 1; ++j) {
                list[j] = list[j + 1];
            }
            list[size - 1] = nullptr;
            --size;
            cout << target << " has been removed from the list.\n";
            return;
        }
    }
    cout << target << " not found in the list.\n";
}

int main() {
    const int MAX_ITEMS = 10;
    GroceryItem* GroceryList[MAX_ITEMS];
    int itemCount = 0;

    GroceryList[itemCount++] = new Fruit("Apple", 10, 7);
    GroceryList[itemCount++] = new Fruit("Banana", 10, 8);
    GroceryList[itemCount++] = new Vegetable("Broccoli", 60, 12);
    GroceryList[itemCount++] = new Vegetable("Lettuce", 50, 10);

    cout << "\n--- Grocery List ---\n";
    for (int i = 0; i < itemCount; ++i) {
        GroceryList[i]->printDetails();
    }

    cout << "\nTotal before deletion: ₱" << TotalSum(GroceryList, itemCount) << endl;

    DeleteItem(GroceryList, itemCount, "Lettuce");

    cout << "\n--- Updated Grocery List ---\n";
    for (int i = 0; i < itemCount; ++i) {
        GroceryList[i]->printDetails();
    }

    cout << "\nTotal after deletion: ₱" << TotalSum(GroceryList, itemCount) << endl;


    for (int i = 0; i < itemCount; ++i) {
        delete GroceryList[i];
    }

    return 0;
}
```

The Output:

8. Conclusion

Provide the following:

Summary of lessons learned
Analysis of the procedure
Analysis of the supplementary activity
Concluding statement / Feedback: How well did you think you did in this activity? What are your areas for improvement?

Conclusion

If there's anything this activity drilled into my head, it's that C++ doesn't really let you coast. You actually have to understand how memory works, like why destructors matter or what the point is of having a copy constructor at all. I kinda realized that you can't just make things work and call it a day. You have to think ahead, especially when you're dealing with pointers and dynamic arrays. It's not just coding, it's like... housekeeping for your own logic.

The whole process honestly took more out of me than I expected. At first, making a class for fruits and vegetables felt basic, like alright, just throw in some attributes, call a constructor, easy. But once I had

to manage memory properly, actually make copies of objects the right way, and deal with deletion without crashing everything, it hit me that there's so much going on behind the scenes. I had to slow down, double-check how I was passing things around, and whether I was accidentally creating leaks.

The activity was clever though. It looked simple, just a grocery list, but it was hiding all these important C++ concepts in it. It forced me to actually implement what I know instead of just memorizing it. Like when I had to delete the lettuce, I literally paused and went, wait, this is where I make sure my destructor works. That's when I started appreciating how everything fits together. It wasn't about just printing results anymore, it was about writing code that could survive test cases and mistakes.

I think I did okay. Not perfect. I second-guessed myself a lot, especially with the syntax and how to organize the objects inside the array. But I understood the point. I didn't just copy-paste my way through. I had to actually reflect and clean up the logic I wrote. There's a lot of space to grow still, especially in handling memory better and keeping my code cleaner. But I definitely feel more confident after this.

So yeah, in the end, I'm walking away from this knowing I've learned something real. Not just how to solve the problem, but how to write code that's aware of its own impact. And that honestly makes the effort worth it.

9. Assessment Rubric