ACTIVITY NO. 2

ARRAYS, POINTERS AND DYNAMIC MEMORY ALLOCATION	
Course Code: CPE010	**Program:** Computer Engineering
Course Title: Data Structures and Algorithms	**Date Performed: 08/04/25**
Section: CPE21S4	**Date Submitted: 08/04/25**
Name: Cruz, Axl Waltz E.	**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;
```

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

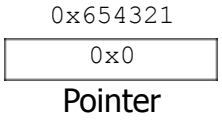
```
type *var_name
int *intPtr
double *dbPtr
```

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;
```

<u>Part C: Dynamic Memory Allocation</u>

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

Memory

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

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;
```

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];
```

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

```
array[index];
```

But initialization is different:

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

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
```

Alternatively, we can use the member access operator:

```
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

Table 2-1. Initial Driver Program

This code use direct initialization which is the student1("Roman", 28) followed by copy constructor is called called when creating student2(student1) and then copy assignment that use in student3 = student2; so the destructor is called for all 3 objects automatically at the end.

main.cpp

Share Run

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

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

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

    // Destructor
    ~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;
    }
};

int main() {
    Student student1("Roman", 28);
    Student student2(student1);
    Student student3;
    student3 = student2;
    return 0;
}
```

Output

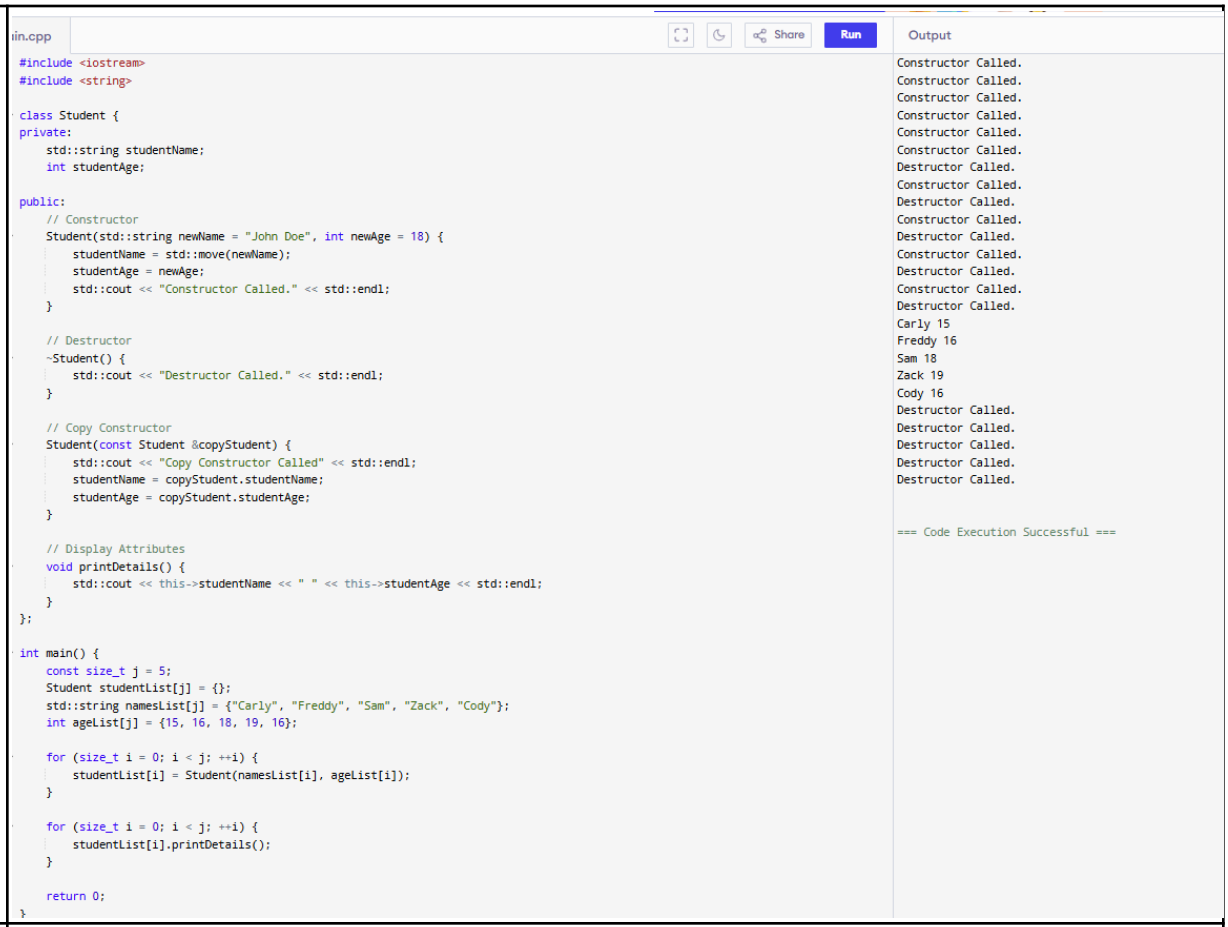
```
Constructor Called.
Copy Constructor Called
Constructor Called.
Destructor Called.
Destructor Called.
Destructor Called.


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

Table 2-2. Modified Driver Program with Student Lists

This code's default constructor called 5 elements in studentList [j], each element is assigned a new temporary student object using copy assignment, all objects are destroyed automatically at the end and no memory leaks which everything is stack allocated.

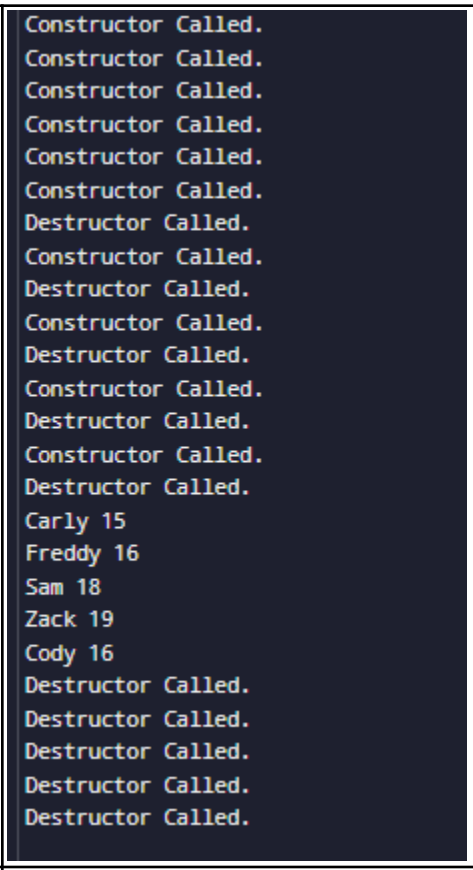
Table 2-3. Final Driver Program

Screenshot	
Observation	

Screenshot	
Observation	

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

Observation	Each time it runs, it makes a student then copy it, so constructor and copy constructor both shows up.
Loop B	`for(int i = 0; i < j; i++){ //loop B studentList[i].printDetails();` `}`
Observation	It just prints out the student's name and age to check if they're set properly.

Output	
Observation	You'll see messages from making, copying, and deleting students, plus their name and age shown.

Modifications	
Observation	The name and age lists are used to give info when making each student.

Table 2-4. Modifications/Corrections Necessary

7. Supplementary Activity

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

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.

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

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

assigned.

Problem 5: Discuss the

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

class Item {
protected:
    string n;
    double p;
    int q;
public:
    Item(string name = "", double price = 0.0, int qty = 0) : n(name), p(price), q(qty) {}
    virtual ~Item() {}
    virtual void show() const = 0;
    virtual double total() const = 0;
    virtual string getName() const = 0;
    virtual Item* copy() const = 0;
};

class FruitItem : public Item {
public:
    FruitItem(string name = "", double price = 0.0, int qty = 0) : Item(name, price, qty) {
        cout << "Fruit made\n";
    }
    ~FruitItem() {
        cout << "Fruit gone: " << n << endl;
    }
    FruitItem(const FruitItem& other) : Item(other.n, other.p, other.q) {
        cout << "Fruit copy\n";
    }
    FruitItem& operator=(const FruitItem& other) {
        if (this != &other) {
            n = other.n;
            p = other.p;
            q = other.q;
        }
        return *this;
    }
    void show() const override {
        cout << "Fruit: " << n << ", $" << p << ", x" << q << endl;
    }
    double total() const override {
        return p * q;
    }
    string getName() const override {
        return n;
    }
    Item* copy() const override {
        return new FruitItem(*this);
    }
};
```

```
Fruit made
Veg made
Fruit made
Veg made
Veg made

List:
Fruit: apple, $1.5, x4
Veg: carrot, $0.9, x5
Fruit: banana, $1.2, x6
Veg: lettuce, $2, x1
Veg: tomato, $1, x3

Total: $22.7
Veg gone: lettuce
Lettuce removed.
Fruit gone: apple
Veg gone: carrot
Fruit gone: banana
Veg gone: tomato


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

```cpp
class VegItem : public Item {
public:
    VegItem(string name = "", double price = 0.0, int qty = 0) : Item(name, price, qty) {
        cout << "Veg made\n";
    }
    ~VegItem() {
        cout << "Veg gone: " << n << endl;
    }
    VegItem(const VegItem& other) : Item(other.n, other.p, other.q) {
        cout << "Veg copy\n";
    }
    VegItem& operator=(const VegItem& other) {
        if (this != &other) {
            n = other.n;
            p = other.p;
            q = other.q;
        }
        return *this;
    }
    void show() const override {
        cout << "Veg: " << n << ", $" << p << ", x" << q << endl;
    }
    double total() const override {
        return p * q;
    }
    string getName() const override {
        return n;
    }
    Item* copy() const override {
        return new VegItem(*this);
    }
};

double getTotal(Item* list[], int len) {
    double sum = 0;
    for (int i = 0; i < len; i++) {
        if (list[i] != nullptr)
            sum += list[i]->total();
    }
    return sum;
}

int main() {
    const int size = 5;
    Item* list[size];

    list[0] = new FruitItem("apple", 1.5, 4);
    list[1] = new VegItem("carrot", 0.9, 5);
    list[2] = new FruitItem("banana", 1.2, 6);
    list[3] = new VegItem("lettuce", 2.0, 1);
    list[4] = new VegItem("tomato", 1.0, 3);
```

```
int main() {
    const int size = 5;
    Item* list[size];

    list[0] = new FruitItem("apple", 1.5, 4);
    list[1] = new VegItem("carrot", 0.9, 5);
    list[2] = new FruitItem("banana", 1.2, 6);
    list[3] = new VegItem("lettuce", 2.0, 1);
    list[4] = new VegItem("tomato", 1.0, 3);

    cout << "\nList:\n";
    for (int i = 0; i < size; i++) {
        if (list[i]) list[i]->show();
    }

    cout << "\nTotal: $" << getTotal(list, size) << endl;

    for (int i = 0; i < size; i++) {
        if (list[i] && list[i]->getName() == "lettuce") {
            delete list[i];
            list[i] = nullptr;
            cout << "Lettuce removed.\n";
            break;
        }
    }

    for (int i = 0; i < size; i++) {
        if (list[i]) delete list[i];
    }

    return 0;
}
```

8. Conclusion:

This activity helped me demonstrate the fundamental concept of inheritance polymorphism, construction and dynamic memory management using new and delete. The structure is both flexible and extendable making a solid foundational complex inventory or shopping list application.

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?

9. Assessment Rubric

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