



CL2001 Data Structures Lab	Lab 1 Revision Of Advanced OOP Concepts
---	--

NATIONAL UNIVERSITY OF COMPUTER AND EMERGING SCIENCES

Fall 2025

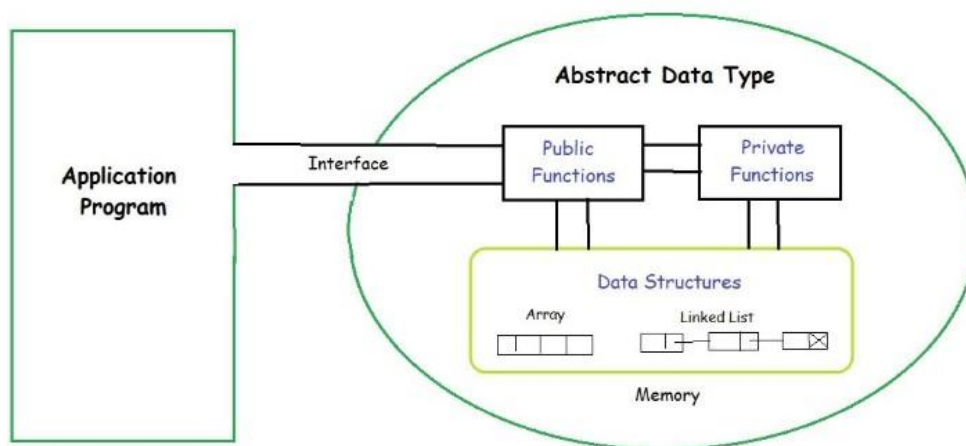
Lab Content

1. Introduction to Abstract Data Types (ADTs)
2. Memory Management in C++ (Pointers, Dynamic Memory Allocation)
3. Constructors & Destructors in C++
4. Rule of Three in C++

Abstract Data Types

Abstract Data Types (ADTs) are theoretical concepts that define the structure and behavior of data. An ADT defines a data structure by the operations that can be performed on it, not by how these operations are implemented. C++ classes are often used to implement ADTs.

This image demonstrates how an Abstract Data Type (ADT) hides internal data structures (like arrays, linked lists) using public and private functions, exposing only a defined interface to the application program.



Pointers in C++

Pointers are a core feature of C++ that allows you to directly manage memory. A pointer holds the memory address of another variable. Understanding pointers is critical for dynamic memory allocation and manipulation of data structures like linked lists and trees.

- **Pointer Basics:**
 - A pointer holds the address of a variable.
 - You can dereference a pointer to access the value stored at the address it points to.



```
-----  
#include <iostream>  
using namespace std;  
  
int main() {  
    int a = 10;  
    int *ptr; // pointer declaration  
    ptr = &a; // storing address of 'a' in 'ptr'  
  
    cout << "Value of a: " << a << endl;  
    cout << "Address of a: " << &a << endl;  
    cout << "Value stored in ptr (address of a): " << ptr << endl;  
    cout << "Value pointed by ptr: " << *ptr << endl; // dereferencing  
  
    return 0;  
}  
-----
```

Dynamic Memory Allocation in C++

In C++, **stack memory** is automatically allocated for variables at compile time and has a fixed size. For greater control and flexibility, dynamic memory allocation on the **heap** is used, allowing manual allocation with **new** and deallocation with **delete**.

Dynamic memory allocation allows you to allocate memory at runtime using the **new** keyword. This is particularly useful for creating data structures where the size is not known at compile time, like linked lists or arrays where the size is determined by the user input.

- **Using new and delete:**
 - **new** allocates memory dynamically.
 - **delete** is used to deallocate memory and prevent memory leaks.

Code for Allocation Of Single Block (Single Variable)

```
-----  
#include <iostream>  
using namespace std;  
  
int main() {  
-----
```



```
// Dynamically allocating memory for a single integer
int* ptr = new int;

// Assigning a value to the dynamically allocated memory
*ptr = 100;

// Displaying the value stored at the allocated memory
cout << "Value of the dynamically allocated integer: " << *ptr <<
endl;

// Freeing the dynamically allocated memory
delete ptr;
return 0;
}
```

Dynamic Memory Allocation for 1D Array

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

int main() {
    int* arr;
    int n;

    // Asking the user for the number of elements
    cout << "Enter the number of elements in the array: ";
    cin >> n;

    // Dynamically allocating memory for an array of integers
    arr = new int[n];

    // Taking input for each element of the array
    for (int i = 0; i < n; i++) {
        cout << "Enter element " << i + 1 << ": ";
        cin >> arr[i];
    }
}
```



```
}

// Displaying the elements of the array
cout << "Array elements: ";
for (int i = 0; i < n; i++) {
    cout << arr[i] << " ";
}
cout << endl;

// Freeing the dynamically allocated memory
delete[] arr;

return 0;
}
```

Deep & Shallow Copy

In general, creating a copy of an object means to create an exact replica of the object having the same literal value, data type, and resources. There are two ways that are used by C++ compilers to create a copy of objects.

- Copy constructor
- Assignment Operator

```
#copy constructor called
Object obj1(obj);
//OR
Object obj1 = obj;

#assignment operator called
Object obj1;
obj1 = obj;
```



In C++, **copying an object** may seem simple at first, but it becomes tricky when the object involves **dynamically allocated memory (DMA)**. When you create a **copy of an object**, you generally have two options:

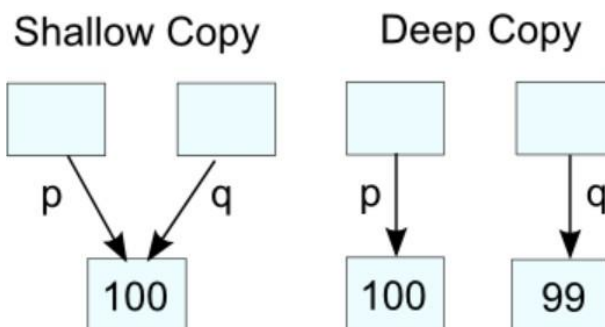
1. **Shallow Copy:** This is the default behavior in C++ when an object is assigned to another object or passed by value. A **shallow copy** means that **only the pointers are copied**, not the actual data. Both the original and the copy will share the same dynamically allocated memory. This can lead to problems such as:
 - **Double-deletion:** Both objects may attempt to free the same memory when they are destroyed, leading to **undefined behavior** (e.g., crashes or memory corruption).
 - **Data corruption:** If one object modifies the dynamically allocated memory, the changes will be reflected in the other object, which might not be the desired behavior.

```
class Box {
public:
    int* size;
    Box(int s) {
        size = new int(s); // Allocate memory dynamically
    }
    ~Box() {
        delete size; // Free the memory
    }
};

Box box1(10);
Box box2 = box1; // Shallow copy: box2.size points to the same memory as box1.size
```

2. **Deep Copy:** A deep copy involves copying not just the pointer, but also the data that the pointer points to. This means creating a new memory block and copying the data from the original object into the new block. After a deep copy, the **original and the copied objects do not share memory**, and thus changes to one object's data do not affect the other.

```
class Box {  
public:  
    int* size;  
    Box(int s) {  
        size = new int(s); // Allocate memory dynamically  
    }  
  
    // Deep copy constructor  
    Box(const Box& other) {  
        size = new int(*other.size); // Create a new memory block and  
        copy data  
    }  
  
    ~Box() {  
        delete size; // Free the memory  
    }  
};  
  
Box box1(10);  
Box box2 = box1; // Deep copy: box2.size gets its own memory
```



Rule of Three in C++

When **Dynamic Memory Allocation (DMA)** is used, we must ensure that our **objects are copied properly** to avoid issues like double-deletion or resource sharing.



The Rule of Three in C++ suggests that **if you define any one of the following three functions**, you should **define all three**. These three functions help ensure proper handling of resources during copying and assignment, especially when DMA is involved:

1. **Destructor**: Cleans up dynamically allocated resources (memory, file handles, etc.) when an object goes out of scope or is explicitly deleted.
 - **Why it's important**: If you don't define a destructor, the **default destructor** may not free the dynamically allocated memory, leading to **memory leaks**.
2. **Copy Constructor**: Creates a **deep copy** of an object when it is passed by value or returned by value.
 - **Why it's important**: Without a custom copy constructor, C++ will perform a **shallow copy** by default, which can lead to **double-deletion** or **data corruption** when objects are destroyed.
3. **Copy Assignment Operator**: Assigns one object's values to another object (already created), ensuring a **deep copy** if necessary.
 - **Why it's important**: Without a custom copy assignment operator, the default **shallow assignment** is performed, which can lead to **resource conflicts**, **double-deletion**, or **incorrect data sharing** between objects.

LAB TASKS

Q1. Create a Student class that stores a student's name and marks for three subjects using DMA. Include functions to set the marks and display the average score. In your program, implement shallow copy by creating one student object, set their marks, and then create another object as a copy of the first. Display the average for both and observe what happens when the original object is destroyed. Then observe the shallow copy problem.

Q2. Create a bank account management system with BankAccount class which should store the balance as a dynamically allocated double. Implement the Rule of Three to manage memory correctly. In your program, create one account, copy it to another, then withdraw some money from the copied account. Display both balances to show that they are independent.

Q3. A travel agency needs a program to manage flight bookings. Create a Flight class that stores a passenger's name (use DMA) and their seat number. Implement the Rule of Three to ensure proper copying and memory management. In your program, create one booking, copy it into another booking, and then modify the name in the copied booking. Display both bookings to confirm that the original remains unchanged.



Q4. Write a program that takes two words from the user. Allocate dynamic memory for both words using `char*`. Then allocate new memory for a concatenated string that contains both words separated by a space. Display the final string and release all allocated memory.

Q5. Create a program that asks the user for the length of their name, allocates memory for it using `char*`, and then takes the name as input. Display the name character by character using pointer arithmetic (without using array indexing). Free the memory after use.