**Encapsulation**

The meaning of **Encapsulation**, is to make sure that "sensitive" data is hidden from users. To achieve this, you must declare class variables/attributes as private (cannot be accessed from outside the class). If you want others to read or modify the value of a private member, you can provide public **get** and **set** methods (getter & setter).

Accessor and mutator are terms used in the context of object-oriented programming, and they are closely related to getters and setters, but they are not exactly the same.

1. **Accessor:**
   * **Definition:** An accessor is a method or function in a class that retrieves the value of a private data member. It provides read-only access to the internal state of an object.
   * **Purpose:** Accessors are used to obtain the values of private member variables without allowing direct modification of those variables from outside the class.
2. **Mutator:**
   * **Definition:** A mutator is a method or function in a class that modifies the value of a private data member. It provides a way to change the internal state of an object.
   * **Purpose:** Mutators are used to modify the values of private member variables in a controlled manner. They allow for the encapsulation of the object's state, ensuring that changes to the state follow specific rules or constraints.
3. **Getter and Setter:**
   * **Getter:** A getter is an accessor method that retrieves the value of a private variable.
   * **Setter:** A setter is a mutator method that modifies the value of a private variable.

**In summary, accessor and mutator are broader terms that encompass the concepts of getting and setting values, respectively. Getters and setters are specific types of accessor and mutator methods, respectively. Accessors provide a way to retrieve information, while mutators provide a way to modify information.**

For example, in your C++ code, the **get** methods like **getID()**, **getName()**, etc., are accessor methods, providing read access to the private member variables. The **set** methods like **setID()**, **setName()**, etc., are mutator methods, allowing you to modify the values of private member variables.

**Shallow Copy**

* A shallow copy creates a new object, but it doesn't create copies of the objects that the original object references.
* In other words, it copies the values of the primitive data members, but for non-primitive data members (like pointers or references), it copies only the memory address (reference), not the actual data pointed to.
* Changes made in one object will affect the other since they share references to the same memory locations.

Shallow copy and pointers – In a shallow copy, two or more pointers of the same type, point to the same memory.

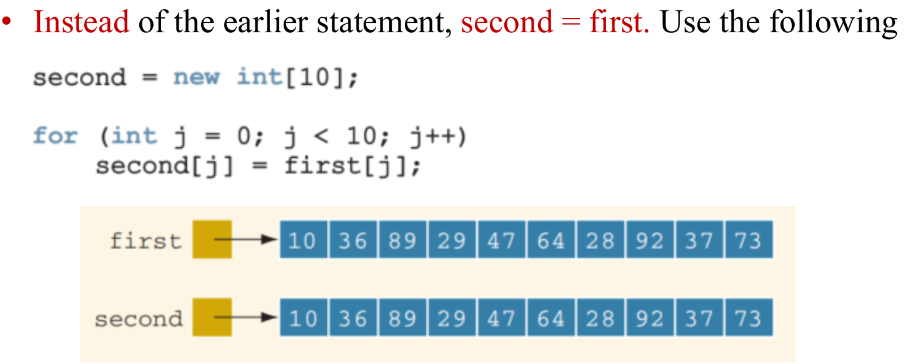
Take 2 pointers, ptr1 and ptr2. Now dynamically allocate memory to ptr1. ptr1 = new int – and initialize ptr1, then ptr2 = ptr1. Now delete ptr2. The first pointer becomes invalid, and both are now **dangling pointers**. Therefore, if the program later tries to access the memory pointed to by first, either the program will access the wrong memory, or it will terminate in an error.

**Deep Copy**

Deep copy refers to the process of creating a separate and independent copy of an object, including all of its member variables or elements. In other words, when you perform a deep copy, you duplicate not only the top-level object but also all of its nested objects or elements.

When you perform a deep copy, you ensure that each object has its own distinct memory allocation. This means that changes made to one object do not affect the other, as they are completely separate entities.

Deep copy and pointer – let’s take the example in shallow



Both first and second now point to their own data. If second deletes its memory, there is no effect on first. In the deep copy, two or more pointers have their own data.

**Static Data Member**

Both regular member functions and static member functions, can access and modify a static data member of the class, but to return the static variable, the member function must be static.

Additionally, constructors and destructors can also access and manipulate the static data members.

Static data members belong to the class itself rather than individual objects of the class. Therefore, they are shared among all instances of the class and can be accessed by any member function, regardless of whether the member function is static or not.

**Static Member Function**

A static member function in C++ is a member function that belongs to the class rather than to any specific object of the class. Here's what a static member function can do and its common use cases:

1. **Access Static Data Members:** A static member function can access and manipulate static data members of the class. Since static data members are shared among all instances of the class, static member functions are often used to perform operations that involve class-wide data.
2. **Access Other Static Member Functions:** Similarly, a static member function can call other static member functions of the class without needing an object instance. This allows for encapsulating class-level functionality that doesn't depend on specific object states.
3. **Access Private Static Members:** Just like non-static member functions, static member functions can access private static data members and other private static member functions of the class. This helps maintain encapsulation and data hiding.
4. **No Access to Non-Static Members:** Unlike regular member functions, static member functions do not have access to non-static (instance) members of the class, including non-static data members and non-static member functions. This is because static member functions do not operate on any specific object instance and can be called without any object.
5. **Counting Instances:** Static member functions are often used to keep track of the number of instances (objects) created for the class. Another method is by incrementing a static data member within the constructor and decrementing it within the destructor, you can count the total number of objects created and destroyed during the program's execution.

**Constant Member Functions**

**No Modifications Allowed**: Inside a constant member function, any attempt to modify the state of the object by changing the values of its data members results in a compilation error. The **const** qualifier acts as a safeguard against unintentional modifications to the object's state.

**Read-only Access**: Constant member functions are often used to provide read-only access to the state of an object. They allow clients of the class to retrieve information about the object's state without the risk of inadvertently modifying it.

**Constant Objects**

**Initialization:** Constant objects must be initialized at the time of declaration because they cannot be assigned a value after initialization.

**Immutable State**: Once initialized, a constant object's state cannot be changed. This means that its data members cannot be modified using member functions or direct assignment.

**Usage Restrictions**: Constant objects can only call member functions that are declared as **const**. This ensures that these functions do not attempt to modify the object's state.

**Compile-time Checks**: Using constant objects enables the compiler to perform compile-time checks to ensure that the object's state is not modified, providing an additional layer of safety and correctness.

**Safety and Predictability**: By declaring objects as **const**, you ensure that their state remains unchanged throughout their lifetime. This can help prevent accidental modifications and lead to more predictable program behavior.



**Dynamic Memory Allocation (DMA)**

Dynamic memory allocation in C++ refers to the process of allocating memory during the runtime of a program. Dynamic memory allocation allows programs to request memory as needed during execution, enabling more flexibility and efficient use of memory resources.

**Areas of Memory**

1. **Stack:**
   * The stack memory is used for storing local variables and function call information.
   * The stack is typically small in size and has a fixed memory allocation determined at compile time.
   * When a function is called, space is allocated on the stack for its local variables and parameters. This space is deallocated when the function returns.
   * The stack is automatically managed by the compiler or runtime system, making it fast and efficient for managing function calls and local variables.
   * Stack memory is limited and can lead to stack overflow errors if it grows too large.
2. **Heap:**
   * The heap is a region of memory used for dynamic memory allocation.
   * It is larger than the stack and can grow and shrink dynamically during the execution of a program (runtime).
   * Memory allocated on the heap persists until explicitly deallocated by the programmer.
   * Dynamic memory allocation functions like **malloc()** in C or **new** in C++ are used to allocate memory on the heap.
   * Unlike the stack, memory on the heap is not automatically managed, and it is the programmer's responsibility to deallocate memory when it is no longer needed to avoid memory leaks.
   * Heap memory is slower to allocate and deallocate compared to stack memory, as it involves more complex memory management operations.

**Memory Leak**

In C++, a memory leak occurs when a program fails to deallocate memory that it has previously allocated dynamically. This means the memory that was allocated using functions like **new** or **malloc()** is not properly released using corresponding deallocation functions **delete** or **free()**, leading to a gradual consumption of system memory over time.

**OR**

**Definition and Cause:** A memory leak in C++ occurs when a program allocates dynamic memory (heap memory) using operators like **new** or functions like **malloc()**, but fails to deallocate it with **delete** or **free()** respectively. As a result, the memory remains allocated and inaccessible, leading to wasted resources.

Memory leaks can degrade the performance of a program and can eventually lead to the program crashing due to running out of memory. They are particularly problematic in long-running applications, such as servers or daemons, where the program continues to allocate memory but fails to release it.

**OR**

**Consequences:**

* **Performance Degradation:** Over time, memory leaks can cause the application to slow down as the available memory for new allocations decreases.
* **System Instability:** In severe cases, it can lead to program crashes or system instability due to the exhaustion of memory resources.
* **Resource Waste:** Memory leaks waste system resources, affecting not only the leaking application but also other applications running on the same system.

To avoid memory leaks, it's essential to ensure that memory allocated dynamically is properly deallocated when it's no longer needed. This can be done by following best practices such as:

**Prevention Strategies:**

* **Proper Deallocation:** Ensure that for every memory allocation, there is a corresponding deallocation.
* **RAII (Resource Acquisition Is Initialization):** Utilize RAII principles where object lifetimes automatically manage resources, ensuring that resources are released when they are no longer needed.
* **Smart Pointers:** Use smart pointers (e.g., **std::unique\_ptr**, **std::shared\_ptr**) provided by the C++ Standard Library, which automatically manage memory and help prevent leaks.
* **Standard Library Containers:** Prefer using containers like **std::vector**, **std::map**, etc., which manage their memory and reduce the risk of leaks.
* **Memory Leak Detection Tools:** Utilize tools such as Valgrind, Visual Leak Detector, or built-in tools in development environments to detect and fix memory leaks during development.

**Key Terms:**

* **Dynamic Memory Allocation:** Allocating memory at runtime using **new** and **malloc()**.
* **Deallocation:** Releasing allocated memory back to the system using **delete** or **free()**.

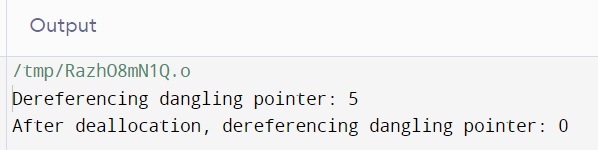
**Dangling Pointer**

Dangling pointers refer to pointers that continue to point to memory that has been deallocated, leading to undefined behavior when dereferenced. This typically occurs when the memory that a pointer is pointing to is freed or deleted, but the pointer is not updated to reflect this change.

When memory is deallocated (e.g., using **delete** in C++), the pointer pointing to that memory becomes a dangling pointer because it still holds the address of the deallocated memory. Dereferencing such a pointer can lead to unexpected behavior, as the memory it points to may have been reallocated for other purposes or may no longer be accessible.

A screenshot of a computer program

Description automatically generated



**Polymorphism**

Polymorphism in C++ means, the same entity (function or object) behaves differently in different scenarios. Polymorphism simplifies code maintenance and promotes code reuse and flexibility.

Polymorphism is a core concept in object-oriented programming (OOP) that allows objects of different classes to be treated as objects of a common superclass. It enables a single interface to represent multiple types of objects.

A single function is used to perform many tasks with the same name and different types of arguments. It is an example of compile-time polymorphism. Another example of compile-time polymorphism is Operator overloading, means defining additional tasks to operators without changing its actual meaning.

Function overriding is a part of runtime polymorphism. In function overriding, more than one method has the same name with different types of the parameter list.

**Virtual Base Class**

A virtual base class is a base class that serves as a common ancestor for multiple derived classes within an inheritance hierarchy. The key feature of a virtual base class is that it enables the sharing of common base class members among the derived classes, ensuring that only one instance of the shared base class exists in memory, even if it's inherited by multiple paths in the hierarchy.

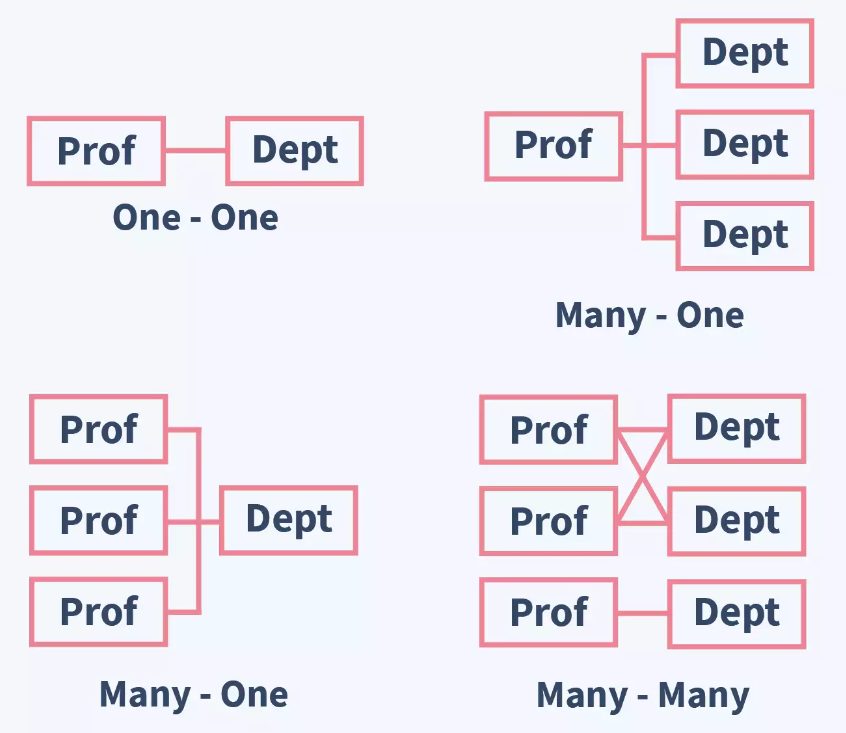
This mechanism is particularly useful in scenarios involving multiple inheritance, especially in cases like the diamond inheritance problem, where a derived class inherits from two or more classes that share a common base class. Without virtual inheritance, this can lead to ambiguity in member access from the common base class.

OR

A virtual base class is a class that is used as a base class for multiple derived classes in a hierarchy, and it enables the prevention of ambiguity issues that can arise from multiple inheritance. When a base class is declared as virtual, it ensures that only one instance of the base class exists in the derived class hierarchy.

**Association**





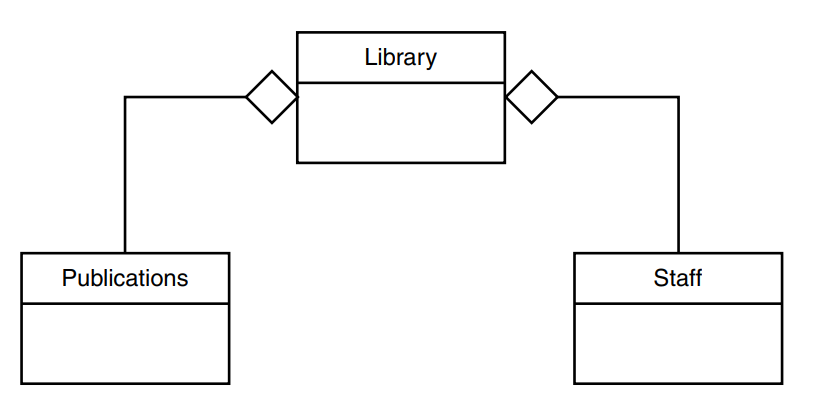
**Aggregation**

If a class B is derived by inheritance from a class A, we can say that “**B is a kind of A.**” This is because B has all the characteristics of A, and in addition some of its own. For this reason, inheritance is often called a “kind of” relationship.

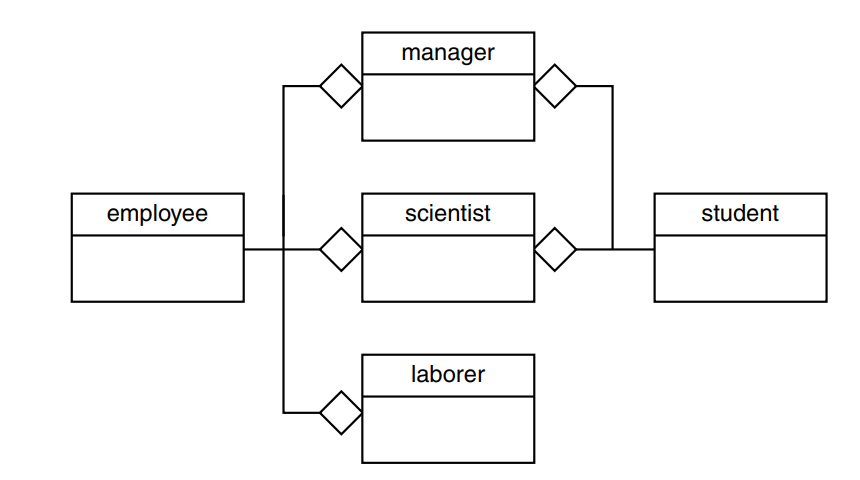
Aggregation is called a “**has a**” relationship. We say a library has a book or an invoice has an item line. Aggregation is also called a “part-whole” relationship: the book is part of the library. In object-oriented programming, aggregation may occur when one object is an attribute of another.

Whole = A class which includes objects of other classes.

UML class diagrams, the “whole” end of the association line has an open diamond-shaped arrowhead.



HAS-a relation: Library has staff and publications **OR** staff and publications are part of the library.

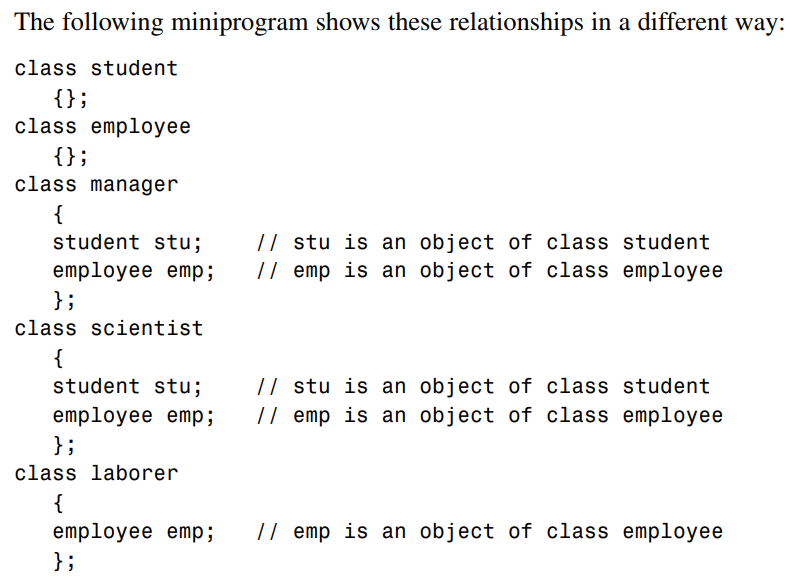


HAS-a relations:

Manager has employee and student **OR** employee and student are part of manager.

Scientist has employee and student **OR** employee and student are part of scientist.

Laborer has employee **OR** employee is a part of laborer.



**Composition**