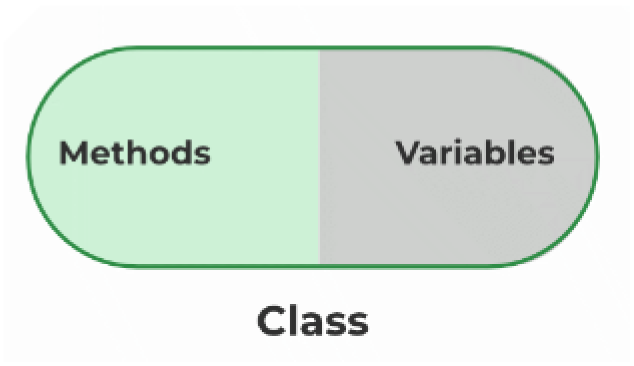
Encapsulation And Constructors

1. Encapsulation

- Encapsulation is defined as the wrapping up of data and information in a single unit.
- In OOP, Encapsulation is defined as binding together the data and the functions that manipulate them.
- In c++ Encapsulation can be achieved using access modifiers and **headers.**

Important Property of Encapsulation

1. **Data Protection:** Protects the internal state of an object by keeping it’s data members private.Access to these is provided to class private members.
2. **Information hiding:** Hiding the implementation details from external code and showing only necessary information.



Access Modifiers

- By default all the data members are made private.
- **Private:** Private members can only be accessed by the same class members.
- **Protected:** Data Members or Function can only be accessed by derived classes or same class members.
- **Public:** Data Members or Functions can be accessed by everyone.

2. Constructors

- Constructors are the special method which are invoked at the time of object creation.
- Constructors have the same name as the class name.
- Constructors should always be public until and unless class is required to be restricted.

Types of Constructors in C++

1. **Default Constructor**
2. **Parameterized Constructor.**
3. **Copy Constructor.**
4. **Move Constructor.**

1. Default Constructor

- Default Constructors are non-static member functions declared with Class name and zero parameters.
- Default constructors are the ones which do not have any arguments. Also called Zero argument constructor.
- If there is no parameterized Constructor defined compiler automatically creates one.

  Syntax:

  ```
  ClassName(){

  }
  ```
- If there is any parameterized constructor present then we need to explicitly create a default constructor. The compiler looks for the argument constructor.if it is there it does not create default constructor.

2. Parameterized Constructor

- *Parameterized Constructors* are non-static member functions declared with Class name and parameters.

  Syntax:

  ```
  ClassName(parameter1, parameter2, ..){
          // set parameters to class fields
          // Perform other constructor operations.
  }
  ```

- Constructor can be called implicitly or explicitly

  ```
  // Explicit call
  ClassName e = ClassName(parameter1, parameter2, ..);
  // Implicit call
  Example ClassName(parameter1, parameter2, ..);
  ```

Default Arguments with C++ Constructor

- Just like the normal function we can define the default value of parameterized Constructor. If that argument is not passed in the constructor default value will be used.

  Syntax:

  ```
  ClassName(int x=0, int b){
          // set parameters to class field
          // Perform other Constructor Operations
  }
  ```

Member Initializer List

- Member initializer List provides a clean and compact way of initializing class parameters using argument Constructor.

  Syntax:

  ```
  ClassName (p1, p2, ...) : mem1(p1),  mem2(p2) ... {
          // set parameters to class field
          // Perform other Constructor Operations
  }
  ```

Constructor Chaining In C++

- Constructor Chaining also known as Delegating Constructor is a way by which one constructor can call the other constructor.
- In C++ we can achieve the Constructor Chaining using member initialization list.The member initializer list is a special syntax used after the constructor's parameter list, where you can specify how the class's data members should be initialized.

  Syntax:

  ```
  // Constructor with parameters
  MyClass(int param1, int param2) : member1(param1),
  member2(param2)
  {
        // Other constructor tasks
  }

  // Constructor with a single parameter (delegating constructor)
  MyClass(int param) : MyClass(param1, param2, param) {
        // Other constructor tasks
  }
  ```

3. Copy Constructor

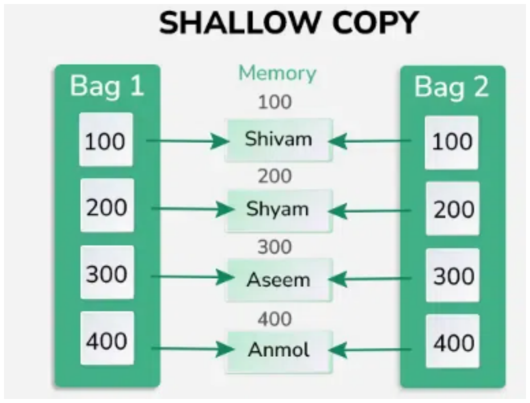
- A copy constructor is an overloaded constructor used to declare and initialize another object.
- Types:
    - Default Copy Constructor
    - User Defined
        - Shallow Copy
        - Deep Copy

Default Copy Constructor

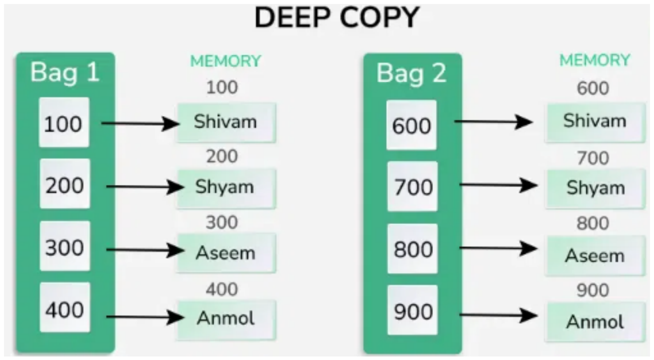
- An implicitly defined copy constructor will copy the bases and members of an object in the same order that a constructor would initialize the bases and members of the object.
- Same as Shallow Copy.

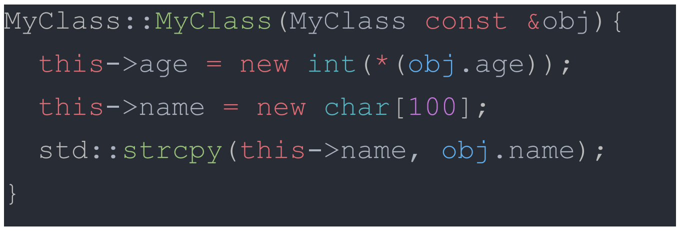
User Defined Copy Constructor

- Shallow Copy
  - **Shallow Copy means that only the pointers will be copied, not the actual resources** that the pointers are pointing to.
  - This can lead to dangling pointers if the original object is deleted.



- Deep Copy
  - **we need to define our deep copy constructor only if an object has pointers or any runtime allocation** of the resource like *a file handle*, a network connection, etc.
  - ***Deep copy is possible only with a user-defined copy constructor.***
  - In a user-defined copy constructor, we make sure that pointers (or references) of copied objects point to a new copy of the dynamic resource allocated manually in the copy constructor using new operators.





```
MyClass::MyClass(MyClass const &obj){
  this->age = new int(*(obj.age));
  this->name = new char[100];
  std::strcpy(this->name, obj.name);
}
```

Move Constructor

- A move constructor is a special type of constructor in C++ that is used to create a new object from the already existing object of the same type, but instead of making a copy of it, it makes the new object point to the already existing object in the memory, leaving the source object in a valid but unspecified state.

- In C++, move constructors is a type of constructor that works on the r-value references and move semantics (move semantics involves pointing to the already existing object in the memory).

- Unlike copy constructors that work with the l-value references and copy semantics(copy semantics means copying the actual data of the object to another object), move constructors transfer the ownership of the already existing object to the new object without making any copy of it.

In C++ there are two types of references-

1. lvalue reference:
    - An lvalue is an expression that will appear on the left-hand side or on the right-hand side of an assignment.
    - Simply, a variable or object that has a name and memory address.
    - It uses one ampersand (&).
2. rvalue reference:
    - An rvalue is an expression that will appear only on the right-hand side of an assignment.
    - A variable or object has only a memory address (temporary objects).
    - It uses two ampersands (&&).

The need or purpose of a move constructor is to steal or move as many resources as it can from the source (original) object, as fast as possible, because the source does not need to have a meaningful value anymore, and/or because it is going to be destroyed in a moment anyway.

```
MoveConstructor::MoveConstructor(MoveConstructor&& obj):age{obj.age}, name{obj.name}{
  std::cout << "Move Constructor Called!!" << std::endl;
  obj.name = nullptr;
  obj.age = nullptr;
}
```