OOP

1. **Introduction to OOP**

   OOP is a way of programming to organize our code. It helps us build systems by thinking about real-world objects and their behaviors.

   Procedural Programming (inline):

   - In procedural programming, we separate data and the operations that work on that data.
   - Sometimes the data is global, making it easy to accidentally modify data from anywhere in the code.
   - Procedural code can be harder to debug and maintain.

   OO Programming:

   - In OOP, we group data and behaviors together in a single object.
   - Data is more secure and can only be accessed through methods.
   - OOP code is easier to debug and organize.

2. **Classes and Objects**

   What is a Class?

   Class is a blueprint or a recipe for how to create an object.

   For Example lets consider the person class:

   to create a class we use the keyword in c++ `class` then followed by the class name

   ```
   class Person{
   }; // notice the semicolon ;
   ```

   - **What is an Object?**

     Object is an instance of a class that contains **both** data and behaviors.

     Objects is made of 2 parts $\begin{cases} Data\ \ attributes \\ Methods \end{cases}$

Data attributes: the variables you add to a class to define and describe the state of the objects from this class.

Methods: the functions/operations that manipulate the data in the class.

Using the previous example:

```
class Person
{
public:
    // Data Attributes
    string name;
    int age;
    double height;
    string job;
    string gender;

    // Methods
    void setName(string n) { name = n; }
    void setAge(int a) { age = a; }
    void setHeight(double h) { height = h; }
    void setJob(string j) { job = j; }
    void setGender(string g) { gender = g; }

    void displayDetails()
    {
        cout << "Name: " << name << endl;
        cout << "Age: " << age << endl;
        cout << "Height: " << height << endl;
        cout << "Job: " << job << endl;
        cout << "Gender: " << gender << endl;
    }

    void celebrateBirthday()
    {
        age++;
        cout << "Happy birthday! You are now " << age << " years old." << end
l;
    }
};
```

- **Create an Object:** To create an object from a class, you declare a variable of the class type. For example:

  ```
  Person person1;  // Object of type Person created
  ```

  Here, `person1` is an object of type `Person` created from the `Person` class.

- **Accessing Data Attributes and Methods:** Once the object is created, you can access the data attributes and methods using the dot `.` operator. For example, to set the name and age of `person1`:

```
person1.setName("Mohamed");
person1.setAge(20);
person1.displayDetails();   // Display the details of the person
person1.celebrateBirthday(); // Increment the age and display a birthday messa
ge
```

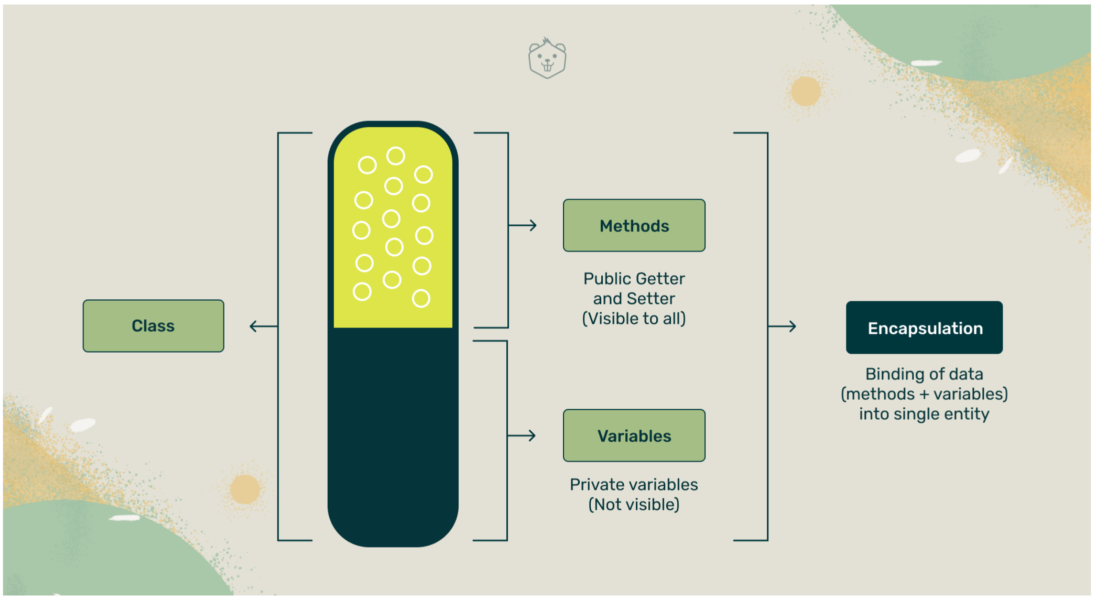
Here, `setName()` and `setAge()` are methods of the `Person` class, and they are called on the `person1` object to set the corresponding attributes.

3. **Encapsulation**

- **what is Encapsulation:**

  Encapsulation is the idea of keeping things together and safe. In OOP, it means bundling data (like variables) and the operations (like functions) that work on that data into a single unit called a class.

  Form it’s name think of Encapsulation as a capsule. The data part is hidden and visible only inside the class, while the methods part is visible even outside the class.


Class
Methods
Public Getter and Setter (Visible to all)
Variables
Private variables (Not visible)
Encapsulation
Binding of data (methods + variables) into single entity

- **Benefits of Encapsulation:**
  - **Data protection:** By hiding the internal data, encapsulation prevents accidental modifications and ensures that the data remains in a valid state.

  - **Code organization:** It helps in structuring the code by grouping related data and functions together, making it easier to understand and maintain.

- **Access Modifiers in C++ Classes:**
  - **Public:** Public members (attributes and methods) are accessible from anywhere in the program. They can be accessed directly using the object of the class.
  - **Private:** Private members are only accessible within the class itself. They cannot be accessed directly from outside the class. To access private attributes, getter and setter methods are used.
  - **Protected:** Protected members are similar to private members, but they can be accessed by derived classes (child classes) as well.

```
class Person
{
private:
    // Private Data Attributes
    string name;
    int age;
    double height;
    string job;
    string gender;

public:
    // public Methods
    void setName(string n) { name = n; }
    string getName() { return name; }

    void setAge(int a) { age = a; }
    int getAge() { return age; }

    void setHeight(double h) { height = h; }
    double getHeight() { return height; }

    void setJob(string j) { job = j; }
    string getJob() { return job; }

    void setGender(string g) { gender = g; }
    string getGender() { return gender; }

    void displayDetails()
    {
        cout << "Name: " << name << endl;
        cout << "Age: " << age << endl;
        cout << "Height: " << height << endl;
        cout << "Job: " << job << endl;
        cout << "Gender: " << gender << endl;
    }

    void celebrateBirthday()
    {
        age++;
        cout << "Happy birthday! You are now " << age << " years old." << end
l;
    }
};
```

Note: By default, members are private if no access modifier is specified.

4. **Constructors and Destructors**

   Constructors:

   - **Purpose:** Constructors are used when creating an object. They initialize the object's data attributes and ensure it is ready to use.
   - **Types of Constructors:**
     - **Default Constructor:** It is automatically called when an object is created without any arguments. It initializes the object with default values or performs specific actions you want to happen when an object is created.
     - **Parameterized Constructor:** It takes parameters during object creation and initializes the data attributes based on these parameters.

   Example:

```
class Person {
private:
    string name;
    int age;
public:
    // Default Constructor
    Person() {
        name = "";
        age = 0;
    }

    // Parameterized Constructor
    Person(string n, int a) {
        name = n;
        age = a;
    }
};
```

   Note: Constructors must be public.

   Destructors:

- **Purpose:** Destructors are called automatically when an object is destroyed (when it goes out of scope). They are responsible for releasing any resources (memory) used by the object.
- **Syntax:**

  A destructor is defined similar to a constructor, but with a tilde `~` before the name.

  Example:

  ```
  class Person {
  public:
      ~Person(){
      }
  };
  ```

  Note: Destructors do not take any parameters and are automatically called when the object is destroyed.

5. **Difference between structs in C and classes**

	Structs in C	**Classes in OOP**
Data and Methods	Structs is used to group related data together. It doesn't have support for member functions or methods.	Classes combines both data and methods. It allows you to bundle behavior (methods) and data together in a single unit.
Encapsulation and Access Control	All members of a struct are public by default, meaning they can be accessed from anywhere in the program.	Class members can have different access specifiers like public, private, and protected. This provides encapsulation, allowing you to control the visibility and accessibility of members.
Constructors and Destructors	Structs don't have constructors or destructors. You need to manually initialize and clean up the struct members.	Classes have constructors and destructors.
Templates	C doesn't have templates, so you can't create generic structs.	OOP supports templates, which enable you to create generic classes.

	Structs in C	Classes in OOP
Inheritance and Polymorphism	Structs don't support inheritance or polymorphism.	Classes support both inheritance and polymorphism