OOPS!!!

- OOP stands for Object-Oriented Programming.

Procedural Programming vs. Object-Oriented Programming (OOP)

- Procedural Programming: Focuses on writing procedures or functions that perform operations on data.

- Object-Oriented Programming: Focuses on creating objects that combine both data and functions into a single unit.

Advantages of OOP Over Procedural Programming

- Efficiency: OOP is faster and easier to execute.

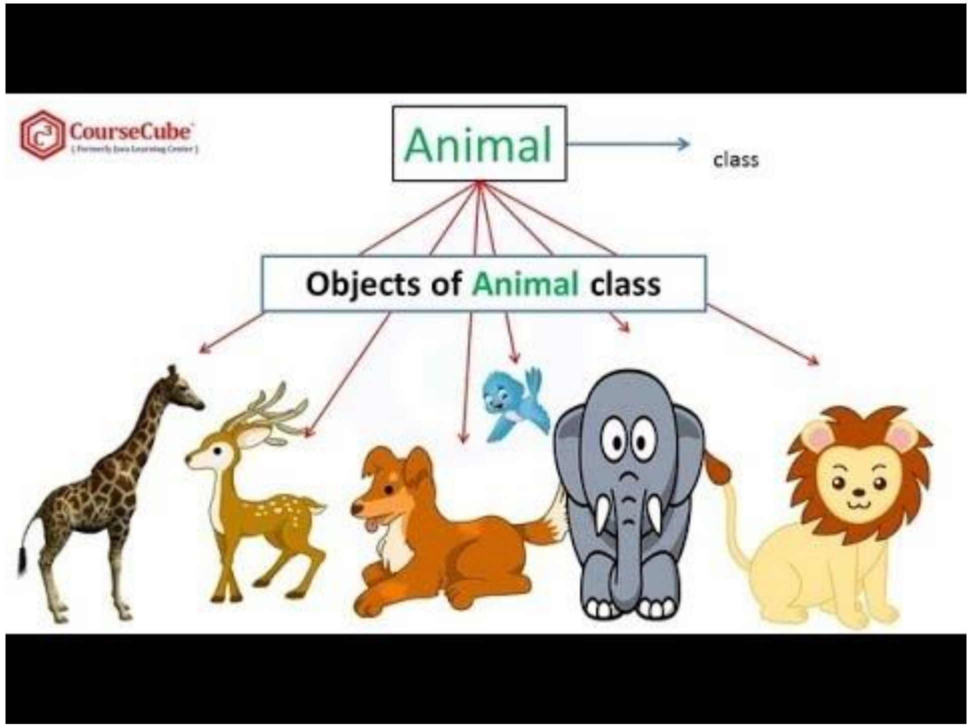
- Clear Structure: OOP provides a well-defined structure for programs, making them more organized.

- DRY Principle: OOP promotes the "Don't Repeat Yourself" (DRY) principle, reducing code repetition and making it easier to maintain, modify, and debug.

- Reusability: OOP allows for the creation of reusable applications with less code and shorter development time.

C++ What are Classes and Objects?

Classes and objects are the two main aspects of object-oriented programming.
Look at the following illustration to see the difference between class and objects:



C++ is an object-oriented programming language.

Everything in C++ is associated with classes, objects, attributes, and methods. For example, in real life, a car is an object. The car has attributes, such as weight and color, and methods, such as drive and brake.

Attributes and methods are variables and functions that belong to the class. These are often referred to as "class members".

A class is a user-defined data type that we can use in our program, and it works as an object constructor or a "blueprint" for creating objects.

Create a class called "MyClass":

```
class MyClass {      // The class
 public:             // Access specifier
  int myNum;         // Attribute (int variable)
  string myString;   // Attribute (string variable)
};
```

Create an Object

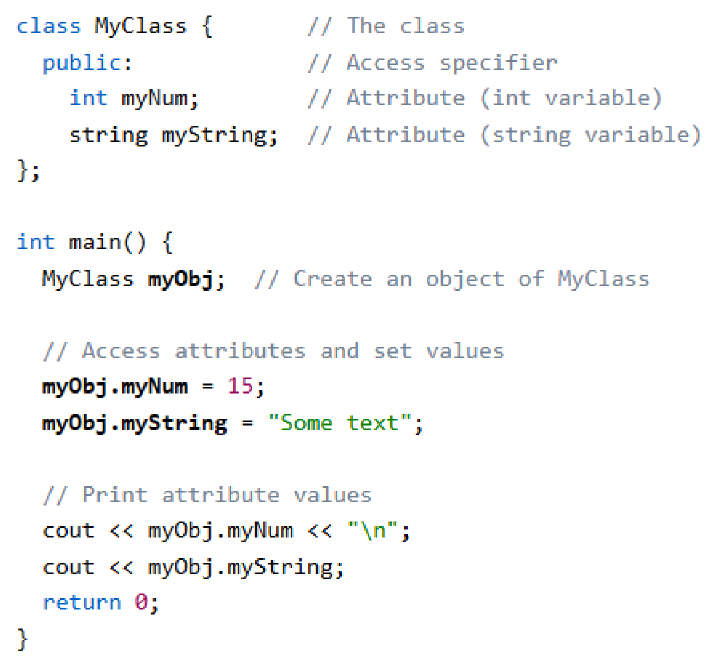
In C++, an object is created from a class. We have already created the class named MyClass, so now we can use this to create objects.

To create an object of MyClass, specify the class name, followed by the object name.

To access the class attributes (myNum and myString), use the dot syntax (.) on the object:

Example

Create an object called "myObj" and access the attributes:



```
class MyClass {       // The class
  public:             // Access specifier
    int myNum;        // Attribute (int variable)
    string myString;  // Attribute (string variable)
};

int main() {
  MyClass myObj;  // Create an object of MyClass

  // Access attributes and set values
  myObj.myNum = 15;
  myObj.myString = "Some text";

  // Print attribute values
  cout << myObj.myNum << "\n";
  cout << myObj.myString;
  return 0;
}
```

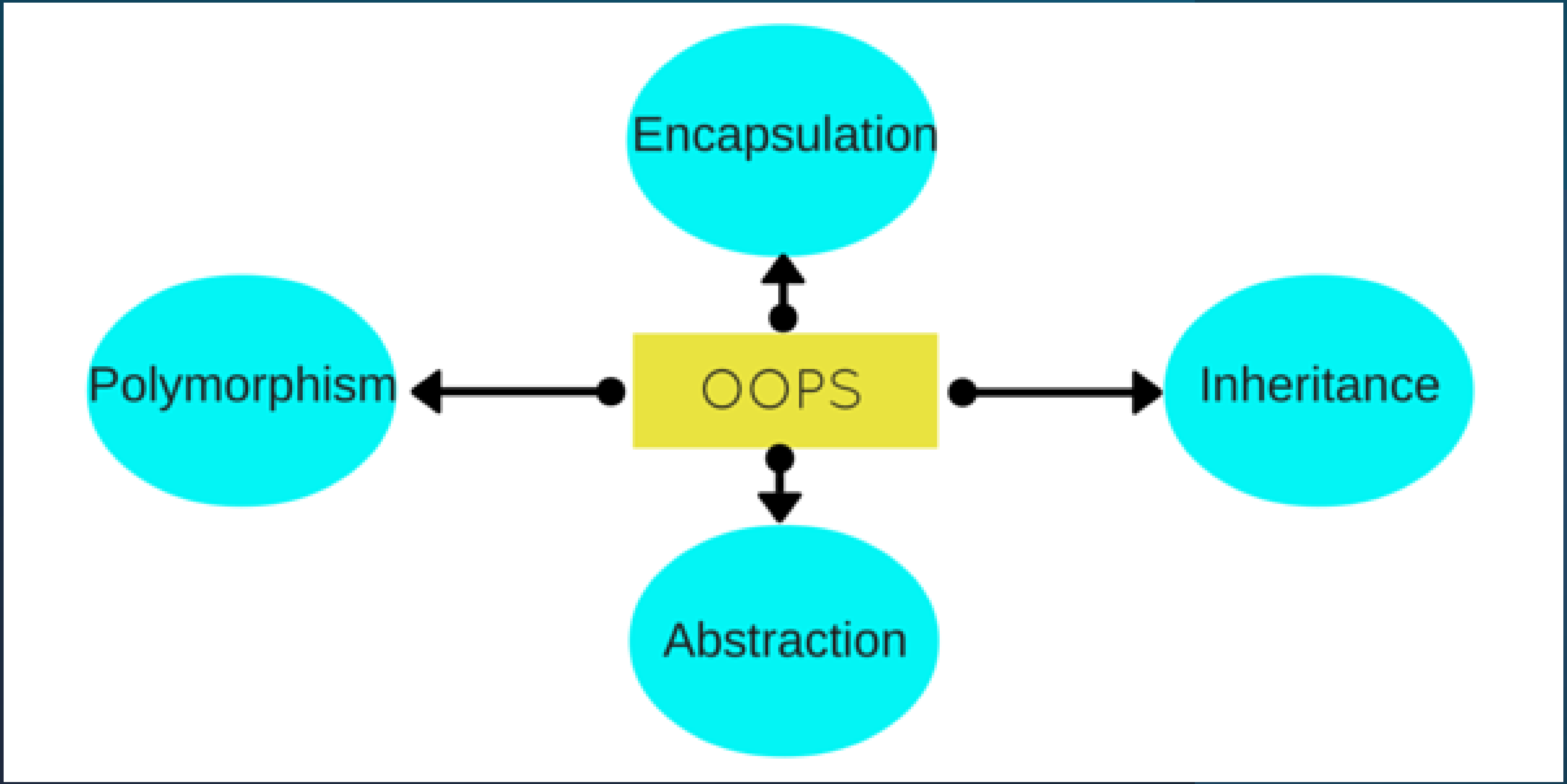
Constructors

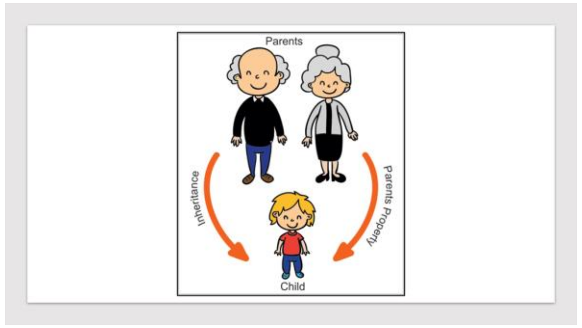
A constructor in C++ is a **special method** that is automatically called when an object of a class is created.

To create a constructor, use the same name as the class, followed by parentheses `()`:

Example

```
class MyClass {     // The class
  public:           // Access specifier
    MyClass() {     // Constructor
      cout << "Hello World!";
    }
};

int main() {
  MyClass myObj;    // Create an object of MyClass (this will call the constructor)
  return 0;
}
```


Encapsulation
Polymorphism
OOPS
Inheritance
Abstraction



Encapsulation – Wrapping data and methods into a single unit (e.g., a **capsule** protecting medicine inside).

Abstraction – Hiding complex details and showing only the necessary features (e.g., **driving a car** without knowing its internal mechanics).

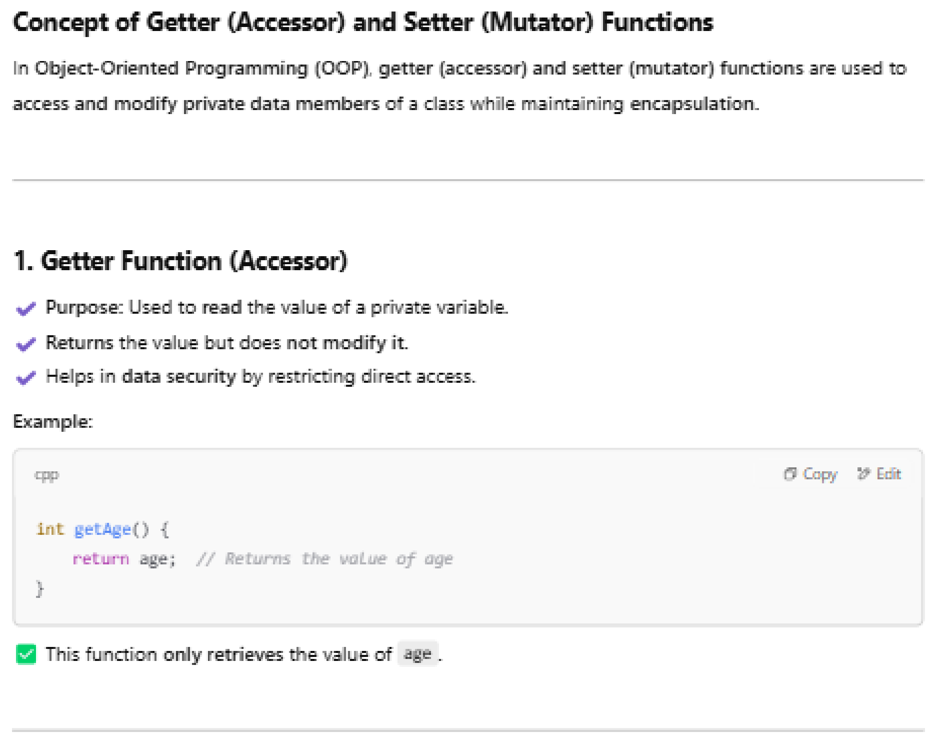
Inheritance – Acquiring properties and behaviors from a parent class (e.g., **a child inheriting** traits from parents).

Polymorphism – The ability to take multiple forms (e.g., **a person** acting as a teacher at work and a parent at home).

getter (accessor) and setter (mutator) functions use both **encapsulation** and **abstraction** in programming.

1. **Encapsulation** – They help **hide data** by keeping variables private. You can only access or change the data using getter and setter methods.
2. **Abstraction** – They **hide the details** of how data is handled. You just use simple methods (get() and set()) without knowing the internal process.

So, they **protect the data** (encapsulation) and **simplify access** (abstraction).



Concept of Getter (Accessor) and Setter (Mutator) Functions

In Object-Oriented Programming (OOP), getter (accessor) and setter (mutator) functions are used to access and modify private data members of a class while maintaining encapsulation.

---

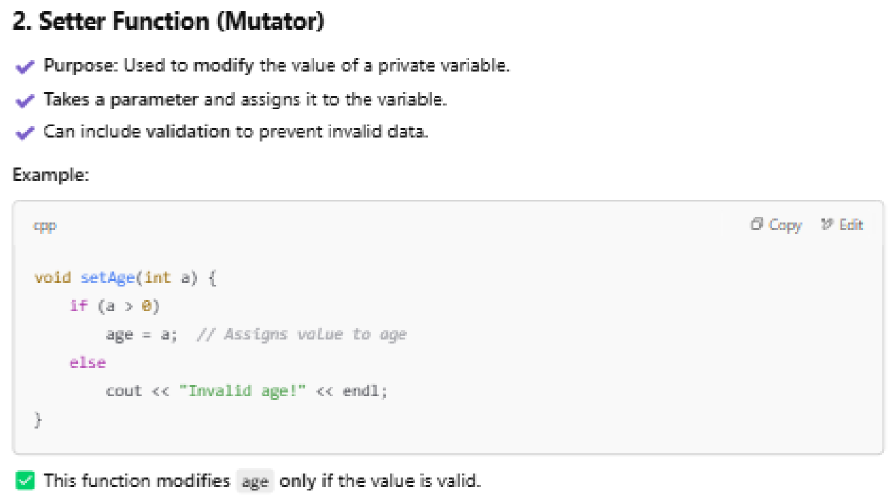
1. Getter Function (Accessor)

✔ Purpose: Used to read the value of a private variable.
✔ Returns the value but does not modify it.
✔ Helps in data security by restricting direct access.

Example:

```cpp
int getAge() {
    return age;  // Returns the value of age
}
```

✅ This function only retrieves the value of `age`.



2. Setter Function (Mutator)

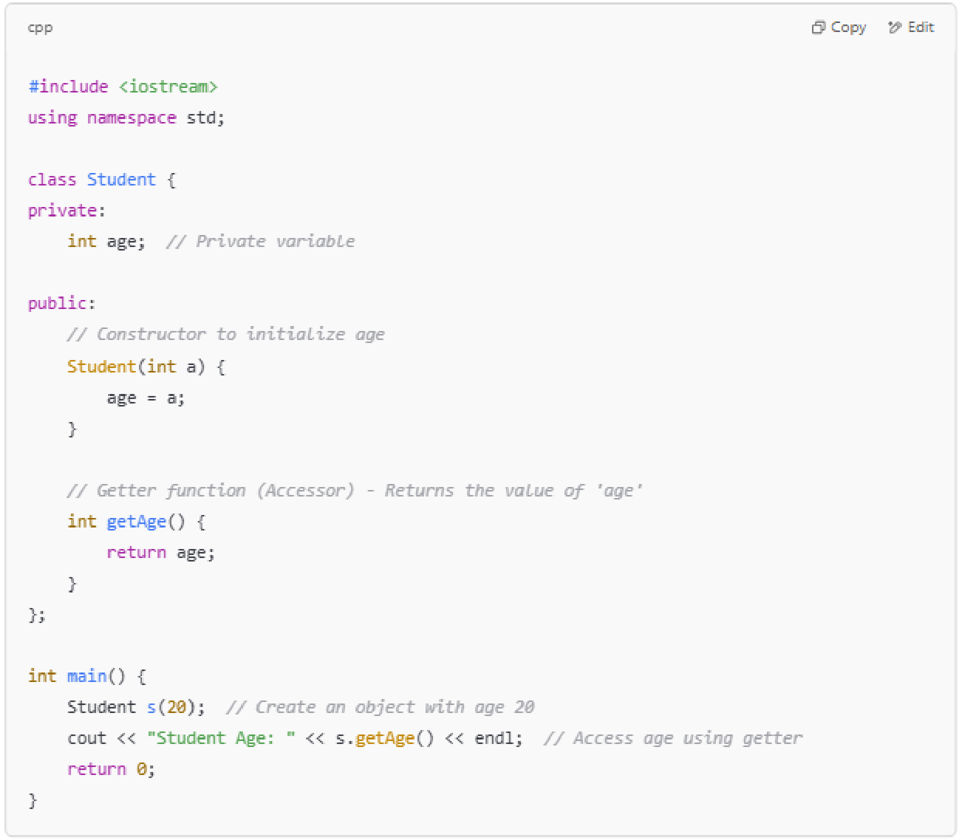
✔ Purpose: Used to modify the value of a private variable.
✔ Takes a parameter and assigns it to the variable.
✔ Can include validation to prevent invalid data.

Example:

```cpp
void setAge(int a) {
    if (a > 0)
        age = a;  // Assigns value to age
    else
        cout << "Invalid age!" << endl;
}
```

✅ This function modifies `age` only if the value is valid.

1. Accessor Function (Getter) - Only Reads Data

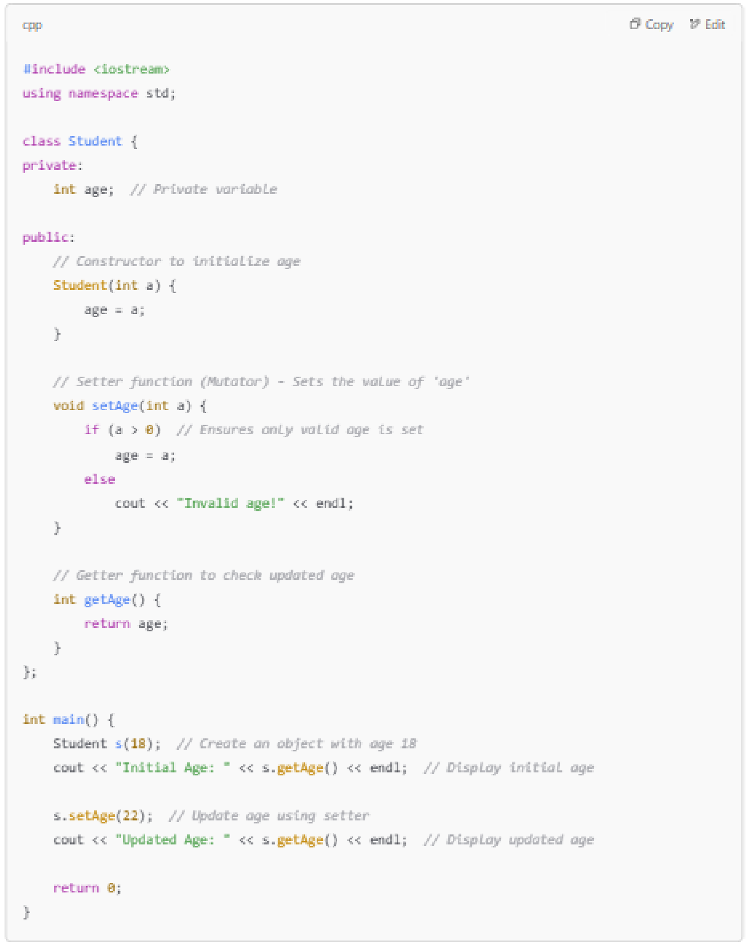


cpp

Copy Edit

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

class Student {
private:
    int age;  // Private variable

public:
    // Constructor to initialize age
    Student(int a) {
        age = a;
    }

    // Getter function (Accessor) - Returns the value of 'age'
    int getAge() {
        return age;
    }
};

int main() {
    Student s(20);  // Create an object with age 20
    cout << "Student Age: " << s.getAge() << endl;  // Access age using getter
    return 0;
}
```

2. Mutator Function (Setter) - Modifies Data



cpp

Copy Edit

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

class Student {
private:
    int age;  // Private variable

public:
    // Constructor to initialize age
    Student(int a) {
        age = a;
    }

    // Setter function (Mutator) - Sets the value of 'age'
    void setAge(int a) {
        if (a > 0)  // Ensures only valid age is set
            age = a;
        else
            cout << "Invalid age!" << endl;
    }

    // Getter function to check updated age
    int getAge() {
        return age;
    }
};

int main() {
    Student s(18);  // Create an object with age 18
    cout << "Initial Age: " << s.getAge() << endl;  // Display initial age

    s.setAge(22);  // Update age using setter
    cout << "Updated Age: " << s.getAge() << endl;  // Display updated age

    return 0;
}
```