SWEG2102 Fall 2026

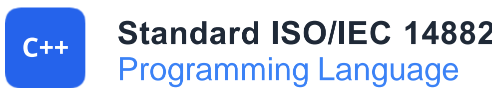
Fundamentals of Programming II

Lecture 07

Basics of Object Oriented Programming (OOP)

Chere Lemma (M.Tech)
Lecturer, Dept. of Software Engineering

Addis Ababa Science and Technology University (AASTU)
Addis Ababa, Ethiopia



Standard ISO/IEC 14882
Programming Language

Outline

Topics Covered

01 What is Object Oriented programming (OOP)?

02 How OOP differ from Procedural Programming?

03 Class and Object Concepts

04 Access Specifiers (Private, Public & Protected)

05 Constructors (Default and Parametrized)

06 Practical Guided Exercise

07 Common Pitfalls & Best Practices

Learning Objectives

By the end of this lecture, you will be able to:

- Define Object-Oriented Programming (OOP) and explain its purpose
- Understand the difference between procedural programming OOP
- Define classes and instantiate objects
- Explain member variables and member functions
- Use access specifiers appropriately and constructors
- Develop a small class-based program using OOP basic concepts
- Identify common mistakes and apply OOP best practices

Understanding
Fundamentals of OOP

What is OOP?

Definition

- **OOP** is a **programming paradigm** that organizes software design around "**objects**," rather than functions and logic.
  - 👉 **Objects** combine data (state) and behavior (methods) into a single unit.
- In OOP we define not only the data types, but also the type of operations/methods (functions) that can be applied to the specified data structure.

Key Idea:

- *The real world can be "accurately" described (modeled) as a collection of objects that interact.*
  - 👉 ***In simple term, model problem domain as interacting objects rather than sequential steps.***

Why OOP?

Problems with Procedural Programming

Limited Data Security

- Data is often **openly accessible** through the program.
- There is **little control** over how data is modified.
- **Sensitive information** can be changed unintentionally.

Tight Coupling of Function

- Functions become **highly dependent** on specific data structures.
- Changes to data structures may requires modifying many functions.
- Small updates can affect multiple parts of the program.

Global/Shared Data Issues

- Global variables create **data dependencies** across the entire program.
- Any function can modify these shared data.
- Changes become difficult to track and debug.

Cont’d

Problems with Procedural Programming

Scalability Problems

- As program features increase, code complexity grows rapidly.
- Adding **new functionality** often needs change in many places.
- This makes difficult to organize and maintain large **procedural code/programs**

Limited code Reusability

- Functions are tied to specific data types or structures.
- **Same logic** might implemented multiple times with slight variations.
- As a result, code is often **duplicated** across different parts of the program.
- This makes code reusability limited.

Difficult to Manage Large Systems

- Large applications become **difficult** to understand & debug.
- Maintenance and collaboration become more challenging.

Cont'd

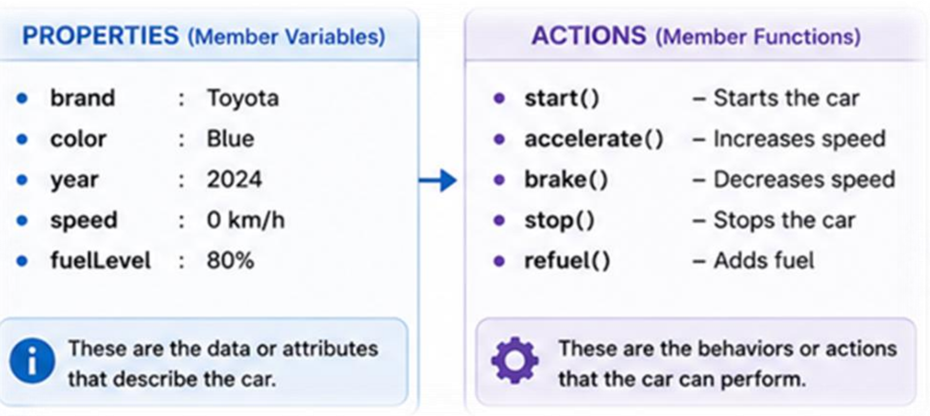
The OOP Solution

OOP addresses these issues by **combine data and behavior** into objects, creating clear boundaries and reducing dependencies between components.

OOP models Real-World Entities

- **OOP** provides a **natural way** to model real-world entities by mapping
  - ***entities → classes***
  - ***properties → member variables***
  - ***actions → member functions***.

Note: ***Real-world Entities*** *are things/concepts in the real-world that we want model in a program.*



Cont'd

Comparison Summary: Procedural vs OOP

PROCEDURAL PROGRAMMING	OBJECT-ORIENTED PROGRAMMING
1. Top-down approach Breaking a problem into procedures, modules, or functions.	**1. Bottom-up approach** Building systems from interacting objects.
2. Focuses on procedures/functions The program is organized around functions.	**2. Focuses on objects** The program is organized around objects.
3. Data and functions are separate Data is often shared globally and functions operate on it.	**3. Data and functions are bundled together** Each object contains its own data (state) and functions (methods).
4. Limited data security No proper mechanism for hiding data; data can be accessed freely.	**4. Supports encapsulation and data hiding** Data is hidden within objects and accessed in a controlled way.
5. Limited or no access specifiers Traditional procedural programming provides limited support for access control.	**5. Supports access specifiers** Provides access specifiers such as private, public, and protected.
6. Difficult to maintain in large systems Best suited for small to medium applications.	**6. Easier to maintain and extend** Designed to handle large and complex systems.
7. Less code re-usability Functions are reused to a limited extent.	**7. Higher code re-usability** Supports re-usability through inheritance and composition.
8. Limited abstraction and flexibility Adding new features may require significant changes.	**8. Strong abstraction and flexibility** Easier to extend functionality with minimal changes.

Fundamental Building Blocks: Class and Object

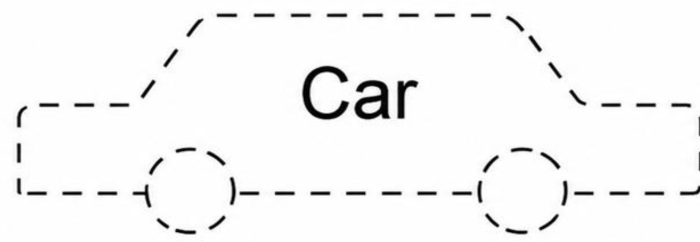
2.1 Class Definition and Syntax

What is Class?

A **class** is a **blueprint** or **template** used to create objects

- Defines the **data** and **behavior** of objects
- Groups **related** variables and functions together
- **Variables** define **attributes (fields/state)**
  - ✓ *Store data about the object*
- **Functions/methods** define **behaviors/actions**
  - ✓ *Define what the object can do*

Class



Properties	**Methods** - behaviors
color	start()
price	backward()
km	forward()
model	stop()

Cont'd

Basic Class Syntax (C++)

```
class className {
    // access specifier: public | private | protected
    public:
        // Data/variable members (attributes)
        dataType variableName;

        // Member functions (methods)
        returnType functionName() {
                // function body goes here
        }
};
```

Date Members (Attributes)

- Variables inside a class.
- Store the **state** of an object.
- By default **protected/hidden**

Member Functions (Methods)

- Functions/methods inside a class.
- Define the **behavior** of the class.
- They can **modify private state** or/and **return** state values.

Cont'd

Example

```cpp
class Circle {
public:
    // The radius of this circle
    double radius;          // Data field

    // Return the area of this circle
    double getArea() {
        return radius * radius * 3.14159;
    }

    // Return the circumference of this circle
    double getCircumference() {
        return 2 * 3.14159 * radius;
    }

    // Return the diameter of this circle
    double getDiameter() {
        return 2 * radius;
    }
```

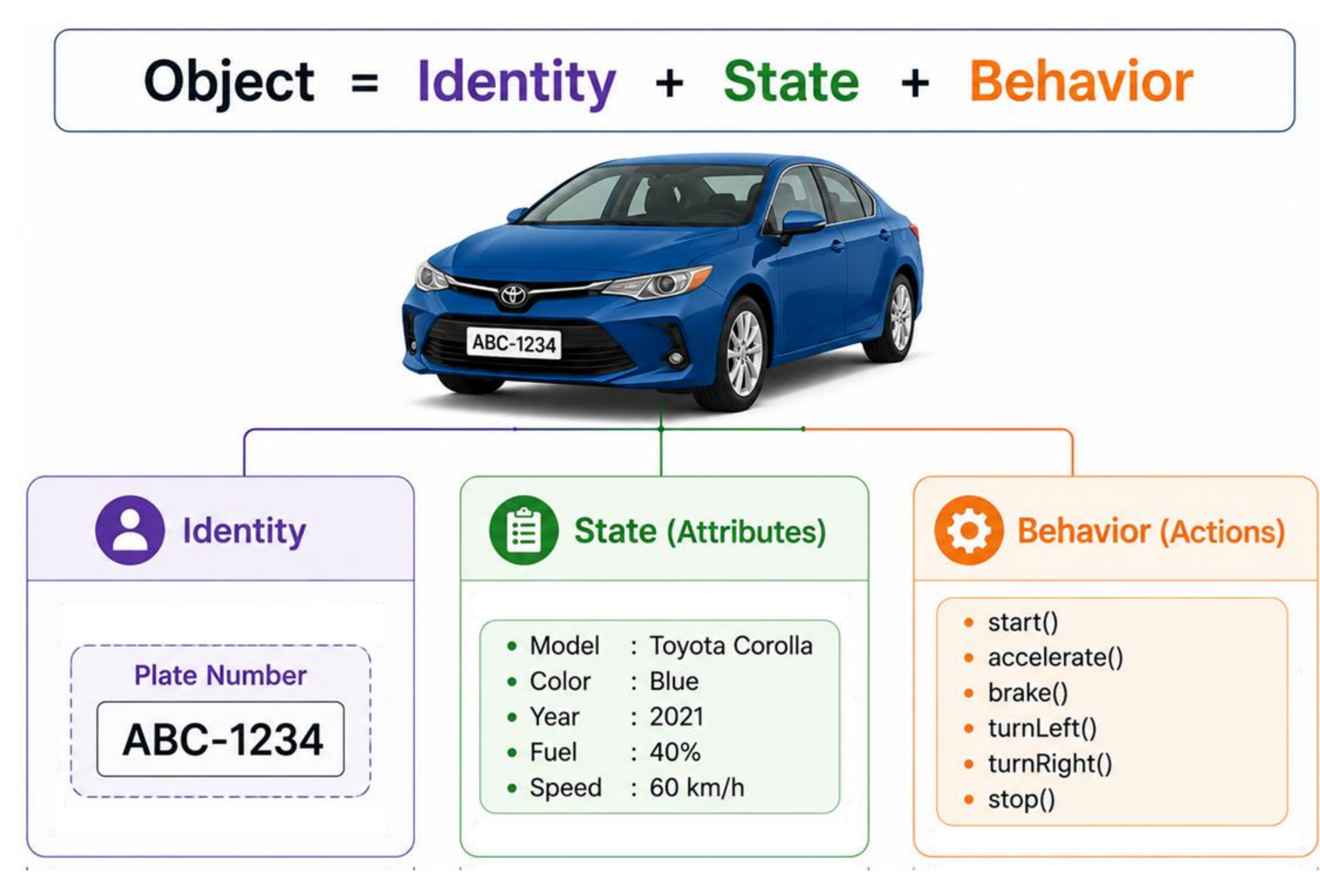
```cpp
    // Display circle information
    void displayInfo() {

        cout << "Radius: "
             << radius << endl;

        cout << "Diameter: "
             << getDiameter() << endl;

        cout << "Circumference: "
             << getCircumference() << endl;

        cout << "Area: "
             << getArea() << endl;
    }
};
```

2.2 Object Creation and Instantiation

What is Object?

An **object** represents the real-world entities

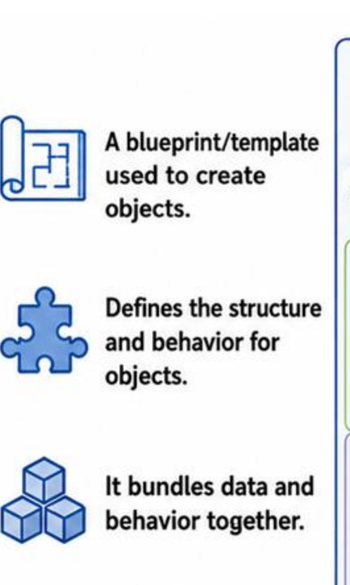
- An instance of a class
- **Object** = **Identity** + **State** + **Behavior**
  - ✓ **Identity** – unique identifier of objects.
  - ✓ **State** – attributes that define current condition of the object.
  - ✓ **Behavior** - represented by actions (methods) of an object

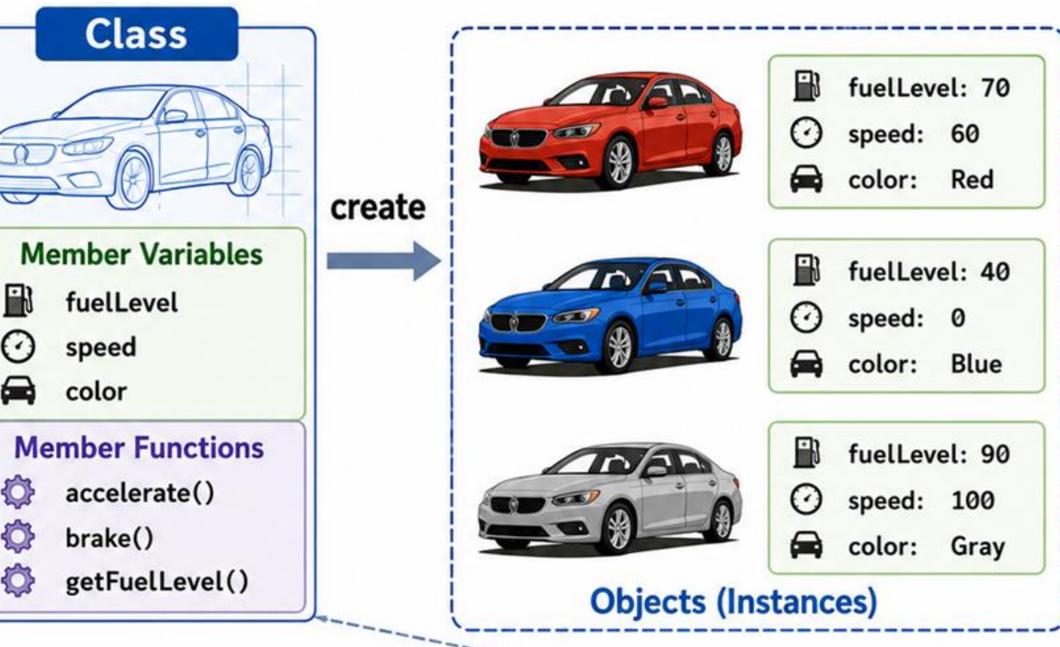


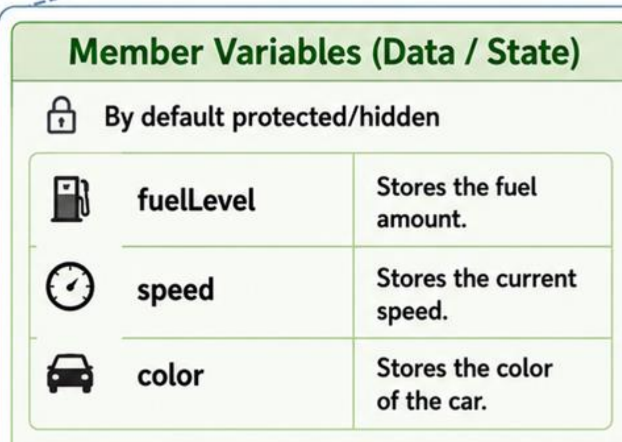
Cont'd

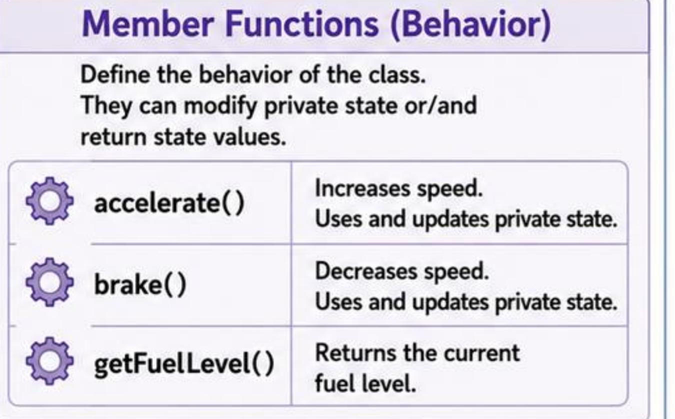
Real-world Analogy:

Putting Together – Class Vs Object









Source:
AI-generated image (ChatGPT 3.5 2026)

Cont'd

Comparison Summary: Class Vs Object

FEATURE	CLASS	OBJECT
Definition	A blueprint or template used to create objects.	A real instance of a class.
Nature	Logical entity – defines structure.	Physical entity – runtime instance.
Purpose	Defines structure and behavior for objects.	Performs operations using class members.
Memory Allocation	Does not occupy memory until objects are created.	Occupies memory when created.
Contain	Attributes and methods definitions.	Actual values and behaviors.
Existence	Exists during program design (compile time).	Exists during program execution (runtime).
Re-usability	Used to create multiple objects.	Represents a specific entity.

Cont’d

Creating Objects/Instances

- **Instantiation** is the process of creating an object from a class.
- **Object instantiation** converts a **class blueprint** into a **real object** stored in memory.
  - ✓ *Class is an **abstract type/user-defined** data type.*
- Each object has **its own data** and can use the class methods (functions).

Objects are typically created by calling a **constructor**, which

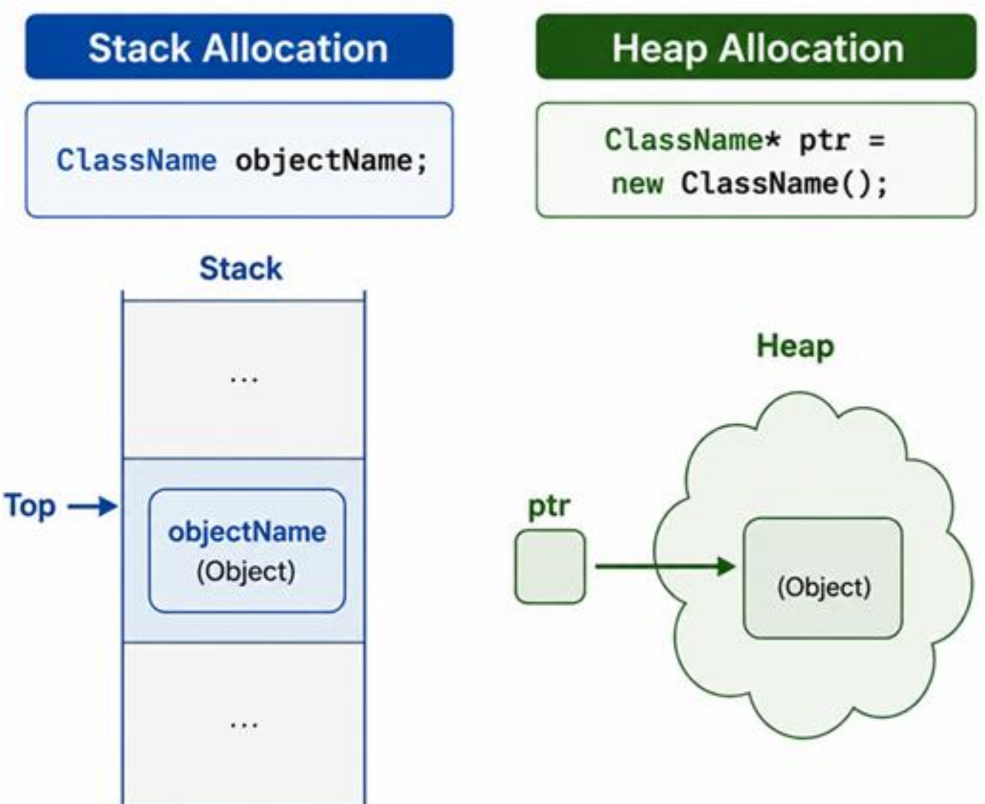
- *Allocates memory for the object*
- *Initializes the object’s data members (properties/attributes)*
- *Prepares the object for use*

Define a **class** → Instantiate the class → **Initialize the object** → Use the object in the program.

Cont’d

Object Creation in C++

- In C++, objects can be created either on the **stack** or on the **heap**.



Example Program

```
class Car {
public:
    void show() {
        cout<<"Car object created"<<endl;
    }
};

int main() {

    // Stack Allocation
    Car car1;
    car1.show();

    // Heap Allocation
    Car* ptr = new Car();
    ptr->show();

    delete ptr;

    return 0;
}
```

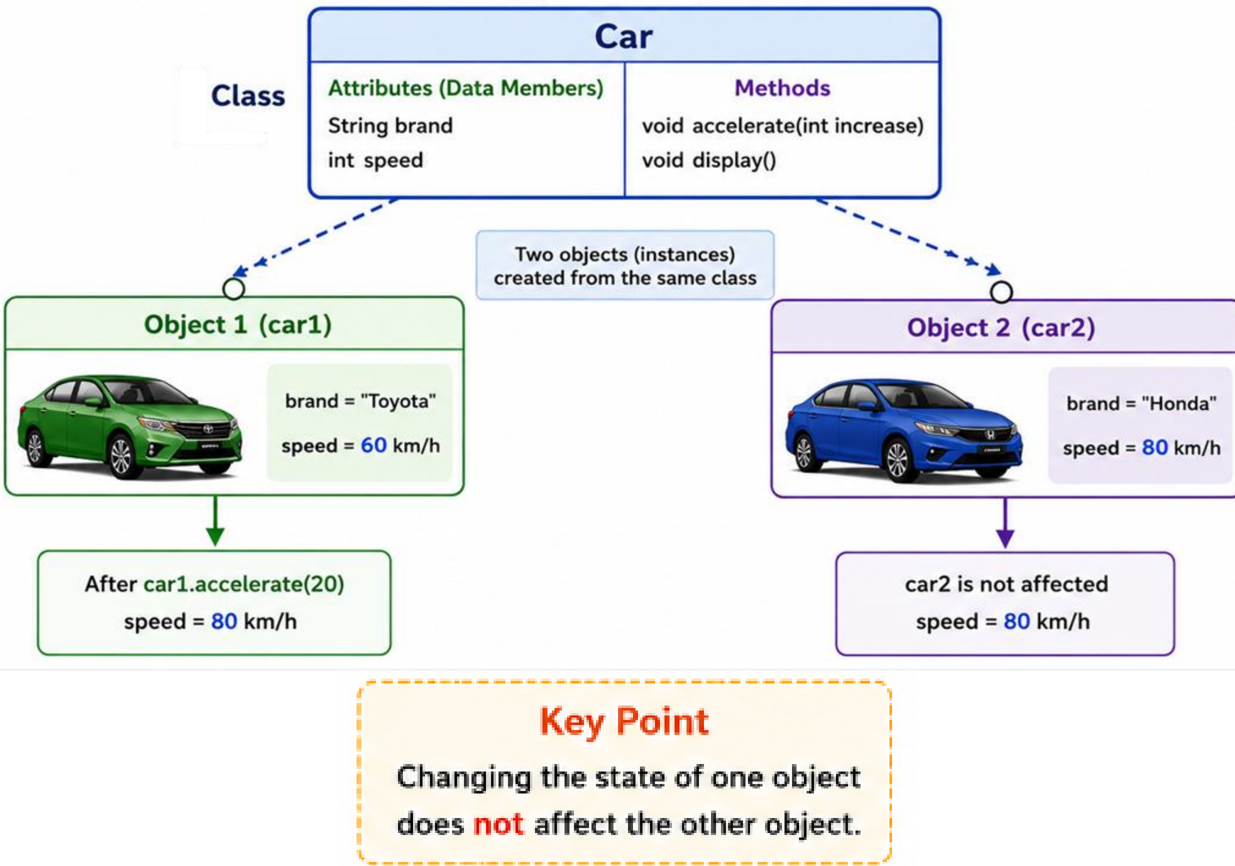
Cont’d

Multiple Objects, Independent State

When you create **multiple objects** from the **same class**, each object **maintains its own independent state**.

```cpp
// Class Definition
class Car {
public:
    string brand;
    int speed;

    void accelerate(int inc) {
        speed += inc;
    }

    void display() {
        cout<<brand <<" : "<<
                speed<<" km/h"<<endl;
    }
};
```

```cpp
int main() {
    // Object 1
    Car car1;
    car1.brand = "Toyota";
    car1.speed = 60;

    // Object 2
    Car car2;
    car2.brand = "Honda";
    car2.speed = 80;

    // Changing only car1
    car1.accelerate(20);

    car1.display();
    car2.display();

    return 0;
}
```



2.3 Access Specifiers

Concepts

- **Access specifiers** also called **access modifiers** define the visibility and accessibility of **class members** (attributes and methods).
- In simple term, define how **class members** can be accessed from outside the class.
- There are **three** access specifiers:
  - ✓ **Private** – accessible only inside the class.
  - ✓ **Public** – accessible from anywhere in the program
  - ✓ **Protected** – accessible inside the class and its derived (child) classes

Why are they Important?

- 🔒 **Protect data** from unauthorized or accidental modification
- 🎯 **Control access (visibility)** to member variables and methods
- 🧩 **Support encapsulation** by hiding internal implementation details
- ♻️ **Improve maintainability** and code organization
- 📦 **Encourages reusability** by protecting against unintended dependencies

Cont'd

Comparison of Access Specifiers in C++

Access Specifier	Accessibility	Accessible Inside Class	Accessible Outside Class	Accessible in Derived Class	Main Purpose
public	Least restricted	Yes	Yes	Yes	Provides general access to members
private	Most restricted	Yes	No	No	Hides and protects data
protected	Restricted inheritance access	Yes	No	Yes	Allows access in derived classes

👉 ***Note:*** *In C++, the* ***default*** *access specifier for a class is* ***private****.*

2.4 Accessing Members

How to access class members?

- The **dot operator (.)** is used to access **public members** of a class object.
- It allows you to:
  - ✓ *Read or modify public member variables*
  - ✓ *Call public member functions*
  - ✓ *Access nested class members*

⚠️ ***Important Note:***

*Private members are not accessible using **dot operator** and it cause compilation error.*

Example Program

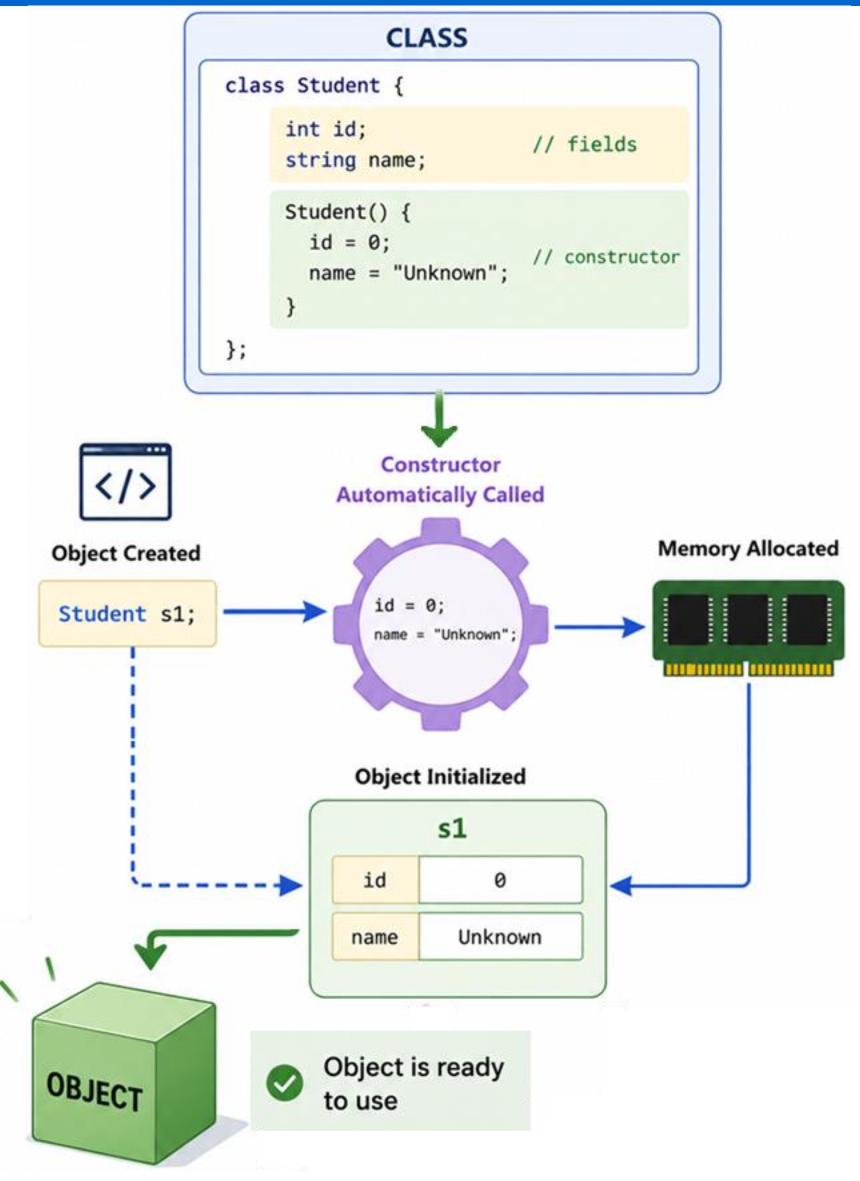
```
class Point {
  public:
    int x; //public member variable
    int y; // public member variable
};

intmain() {
   Point p; // Create a Point object
   // Access x using dot operator
   p.x = 3;
   p.y = 4;
   cout<< p.x <<" "<< p.y<<endl;
  return0;
}
```

Output: 3 4

2.5 Constructors - What and Why?

Concepts

- **Constructors** are **special methods** used in OOP class that is automatically invoked when an object is created.
- They have the **same name** as the class name
- They don't have a **return type**, not even `**void**`.
- Their purpose is to **allocate memory** and **initialize the state** of a new object (assign initial values to its fields).
- Constructors can be overloaded (i.e. multiple constructors with different signatures)



Cont'd

Types of Constructors

1. Default Constructor

- Takes no parameters
- Initializes object with default values
- Called automatically when object is created

```
class Student {
public:
    Student() {
      // initialization code
    }
};
```

2. Parameterized Constructor

- Takes parameters
- Initializes object with specific values
- Allows customized initialization

```
class Student {
public:
    Student(string name, int age) {
      // initialization code
    }
};
```

Real-world Analogy: Buying a Laptop

1 Default Constructor

Buying a laptop with factory/default settings

What you get:
- Standard configuration
- Predefined RAM
- Predefined storage
- Preinstalled software
- No customization

Laptop with factory settings

The object is created with **default values**.

2 Parameterized Constructor

Buying a laptop with custom specifications

You choose:
- Brand / Model
- RAM size
- Storage capacity
- Processor type
- Other features

Customized laptop configuration

The object is created with **user-defined values**.

Example

Complete C++ Example Program

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

// Class Definition
class Student {
private:
    // Private attributes
    int studentId;
    double gpa;

public:
    string name; //Public attributes

    // Default Constructor
    Student() {
        name = "Unknown";
        studentId = 0;
        gpa = 0.0;
        cout<<"Default Constructor Invoked"<<endl;
    }
```

```cpp
    // Parameterized Constructor
    Student(string n, int id, double g) {
        name = n;
        studentId = id;
        gpa = g;
        cout<<"Parameterized Constructor Invoked\n";
    }
    // Public Method
    void displayInfo() {
        cout << "Name: " << name << endl;
        cout << "Student ID: " << studentId << endl;
        cout << "GPA: " << gpa << endl;
    }
    // Setter Method
    void setGPA(double g) {
        if (g >= 0.0 && g <= 4.0)
            gpa = g;
    }
   // Getter Method
   double getGPA() {
      return gpa;
   }
};
```

Cont’d

Complete C++ Example Program

```cpp
int main() {

    // Object Creation on Stack
    Student s1;
    s1.name = "John";
    s1.setGPA(3.5);

    // Accessing members dot operator
    cout << "\n--- Stack Object ---" << endl;
    s1.displayInfo();

    // Parameterized Constructor & Stack Object
    Student s2("Alice", 101, 3.8);
    cout<<"\n--- Stack Object with
            Parameterized Constructor ---\n";
    s2.displayInfo();
```

```cpp
    // Object Creation on Heap
    Student* s3 = new Student("Bob", 202, 3.2);
    cout << "\n--- Heap Object ---" << endl;

    // Accessing members using pointer
    s3->name = "Robert";
    s3->displayInfo();
    cout << "GPA: " << s3->getGPA() << endl;

    // Free heap memory
    delete s3;

    return 0;
}
```

Expected Output

```
Default Constructor Called

--- Stack Object ---
Name: John
Student ID: 0
GPA: 3.5

Parameterized Constructor Called

--- Stack Object with Parameterized
            Constructor ---
Name: Alice
Student ID: 101
GPA: 3.8

Parameterized Constructor Called

--- Heap Object ---
Name: Robert
Student ID: 202
GPA: 3.2
GPA: 3.2
```

Quick Check and Practice Exercise

Quick Check: Pause & Predict

1. **Which is true?**
   A. Object is blueprint/template
   B. Class is instance of object
   C. Object is instance of class
   D. All are incorrect

   Answer: C (Object is instance of class)

2. **What gets called automatically when an object is created?**
   A. A member function
   B. Static method
   C. The constructor
   D. The destructor

   Answer: C (The constructor)

3. **Which is safer and why?**
   A. Exposing balance as a public variable
   B. Modifying it via deposit() or withdraw() methods

   Answer: B (Methods are safer - allow validation. e.g., prevent negative deposits)

4. **In C++, the default access specifier for member of class?**
   A. public
   B. protected
   C. private
   D. None

   Answer: B (c)

Practical Exercise:

Exercise #1— BankAccount Class

Objective

- ➢ Implement a small, safe bank account API to demonstrate.
- ➢ This exercise will **demonstrate** how to:
  - ✓ *Create a class with private member variables*
  - ✓ *Implement constructors for initialization*
  - ✓ *Provide public methods for safe data access*
  - ✓ *Validate inputs to prevent invalid states*

Requirements

1. **Private members:**
   - ✓ accountNumber (string),
   - ✓ balance (double)
2. **Constructors:**
   - ✓ default constructors and
   - ✓ parameterized constructors
3. **Methods:**
   - ✓ deposit(), withdraw(),
   - ✓ getBalance(), getAccountNumber()
4. **Validation:**
   - ✓ reject negative deposits;
   - ✓ prevent overdraft

Cont’d

Solution:

Class Definition and method implementation

```cpp
// Class Definition
class BankAccount {

private:
    string accountNumber; // Account identifier
    double balance;       // Current balance

public:
    // Default constructor
    BankAccount() {
        accountNumber = "0000";
        balance = 0.0;
    }

    // Parameterized constructor
    BankAccount(string acc, double bal) {
        accountNumber = acc;
        balance = (bal < 0) ? 0.0 : bal;
    }
   // Member functions
   void deposit(double amount);
   bool withdraw(double amount);
   double getBalance()
   string getAccountNumber()
};
```

```cpp
// Implementation of deposit method
void BankAccount::deposit(double amount) {
    if (amount > 0) {
        balance += amount; // Add to balance
        cout << "Deposited: $" << amount << "\n";
    } else {
        cout << "Invalid deposit amount\n";
    }
}

// Implementation of withdraw method
bool BankAccount::withdraw(double amount) {
    if (amount > 0 && amount <= balance) {
        balance -= amount; // Subtract from balance
        cout << "Withdrawn: $" << amount << "\n";
        return true;
    } else {
        cout << "Insufficient funds or invalid amount\n";
        return false;
    }
}

// Getter: getBalance
double BankAccount::getBalance() {
    return balance;
}
// Getter: getAccountNumber
string BankAccount::getAccountNumber()  {
    return accountNumber;
}
```

Cont'd

Solution:

Usage inside main() method

```
int main() {

    // Stack Object using Default Constructor
    BankAccount acc1;

    cout << "Default Account Number: "
         << acc1.getAccountNumber() << endl;
    cout << "Initial Balance: $"
         << acc1.getBalance() << endl;

    // Valid method access
    acc1.deposit(150.0);
    acc1.withdraw(50.0);

    // Invalid access to private members
    // we need to comment it or remove it
    acc1.balance = 5000;           // ❌ ERROR
    acc1.accountNumber = "A1";     // ❌ ERROR

    cout << "Updated Balance: $"
         << acc1.getBalance() << endl;
```

```
    // Stack Object using Parameterized Constructor
    BankAccount acc2("B456", 300.0);
    acc2.deposit(100.0);

    bool ok1 = acc2.withdraw(500.0);
    if (!ok1) {
        cout << "Withdrawal failed!\n";
    }

    acc2.withdraw(150.0);
    cout << acc2.getAccountNumber()
         << ": $"
         << acc2.getBalance() << endl;

    // Heap Object using Parameterized Constructor
    BankAccount* acc3 =
        new BankAccount("C789", 500.0);
```

Cont'd

Solution:

Usage inside main() method

```
    // Accessing members using pointer operator
    acc3->deposit(200.0);

    bool ok2 = acc3->withdraw(1000.0);
    if (!ok2) {
        cout << "Insufficient balance!\n";
    }

    cout << acc3->getAccountNumber()
         << ": $"
         << acc3->getBalance() << endl;

    // Free heap memory
    delete acc3;

    return 0;
}
```

Expected Output

```
Default Account Number: 0000
Initial Balance: $0

Deposited: $150
Withdrawn: $50

Updated Balance: $100

Deposited: $100
Insufficient funds or invalid amount
Withdrawal failed!
Withdrawn: $150
B456: $250

Deposited: $200
Insufficient funds or invalid amount
Insufficient balance!
C789: $700
```

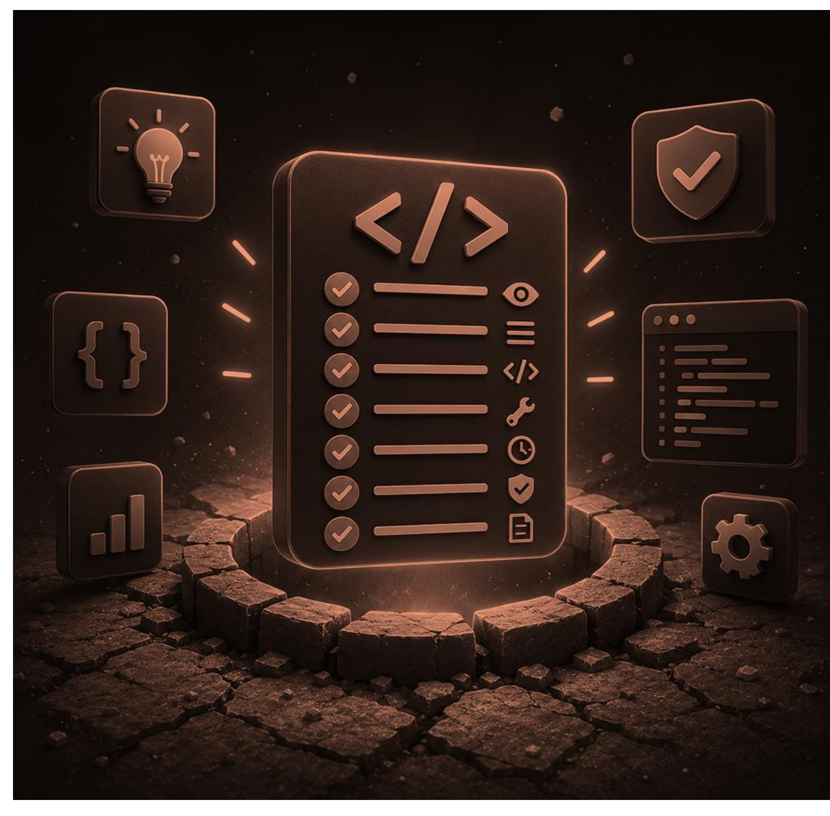
Common Pitfalls and Best Practices

Common Mistakes

- ✕ Using **dot operator (.)** instead of **arrow operator (->)** with object pointers.
- ✕ Accessing **private members** directly outside the class.
- ✕ Using **wrong Access Specifiers** - using public when private is needed, or protected incorrectly.
- ✕ Missing **constructors** and not initializing objects properly.
- ✕ Forgetting to free memory or **delete** objects created on heap using **new** operator.
- ✕ Performing unnecessary heavy operations inside constructors.

Cont’d

Best Practices



- ✓ **Clear class design:** Define classes with single responsibility and clear interfaces - Keep it focused and cohesive
- ✓ **Data protection:** Keep **data members private** and provide **controlled access** through public methods.
- ✓ **Proper Initialization:** Initialize objects to **valid states** using constructors.
- ✓ **Keep Constructors Simple:** Avoid heavy computations inside constructors; use them mainly for object initialization.
- ✓ **Consistent Naming**: Use meaningful names for classes, methods, and variables.
- ✓ **Validation in Setters:** Always **validate** inputs before **modifying state** in setter methods.

Summary

Key Takeaways

OOP Fundamentals

- OOP organizes code around **objects** that combine data and behavior.
- It is a paradigm shift from procedural to object-based

Classes & Objects

- **Classes** define blueprints for creating objects.
- **Objects** are instances of a class with their own independent state.
- Multiple objects can be created from the same class

Constructors

- **Initialize** objects to valid states automatically at creation time.
  - ✓ *Default constructor (no params)*
  - ✓ *Parameterized constructor*

Data Protection

- **Access specifiers** protects object integrity
  - ✓ ***Private** members for data hiding*
  - ✓ ***Public** interface for access*
  - ✓ ***Validation** in methods*

References

Text Books

- **C++ How to Program** [10th edition], Deitel, P. & Deitel, H., Global Edition, Global Edition (2017).
- **Problem Solving With C++** [10th edition], Walter Savitch, University of California, San Diego, 2018.

Reference Books

- **Object-Oriented Programming in C++ (4th Edition)** by Robert Lafore, Sams Publishing, 2022.
- **Principles of Object-Oriented Programming**, by Stephen W. & Dung N., OpenStax CNX, 2022.

Online Resources

- https://www.geeksforgeeks.org/cpp/c-plus-plus/
- https://www.w3schools.com/cpp/default.asp
- https://programiz.pro/resources/cpp
- https://www.hackerrank.com/domains/cpp
- https://cplusplus.com/doc/tutorial/

💡 ***Study Tip:*** *Don't just read the code! Retype the examples from these slides and resources into your IDE, modify and compile them to see what happens.*

Thank You!

Addis Ababa Science and Technology University (AASTU)
Addis Ababa, Ethiopia