CS-1201 Object-Oriented Programming

Introduction

Arbish Akram

Department of Computer Science
Government College University

About Course

Code	1201
Title	Object-Oriented Programming (OOP)
Credit Hours	3+1
Semester	Fall 2026
Prerequisite	Programming Fundamentals (PF)

Recommended Books

1. **Object-Oriented Programming in C++** by Robert Lafore, 4th Edition
2. **C++ How to Program** by Paul J. Deitel and Harvey Deitel, 8th Edition
3. **Starting Out with C++** by Tony Gaddis, 8th Edition

Quizzes

- Quizzes can be announced or unannounced.
- There is no retake for any quiz.

Assignments

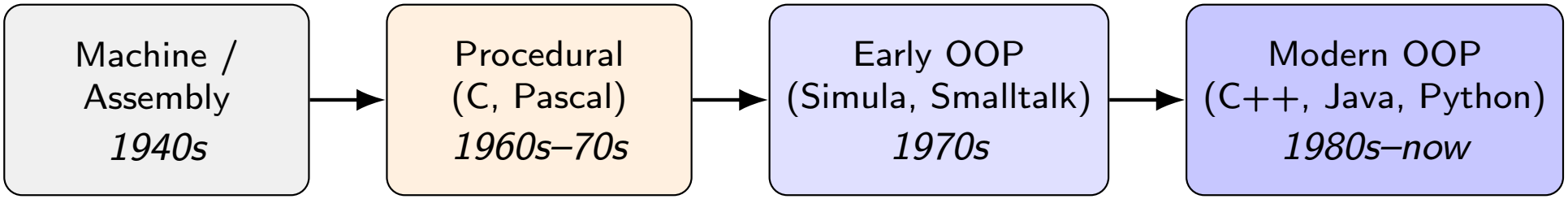
- Instructions in each assignment must be followed carefully.
- Failure to follow instructions will result in a score of 0.
- Deadlines for assignments will not be extended.

Attendance

- Maintaining at least 80% attendance is required to be eligible for the final exam.
- No favors regarding attendance will be granted.
- If you miss attendance during roll call, it will not be marked again.

From Machine Code to Objects

- Programming languages evolved as programs grew larger and harder to manage.
- Each new paradigm solved problems the previous one could not.

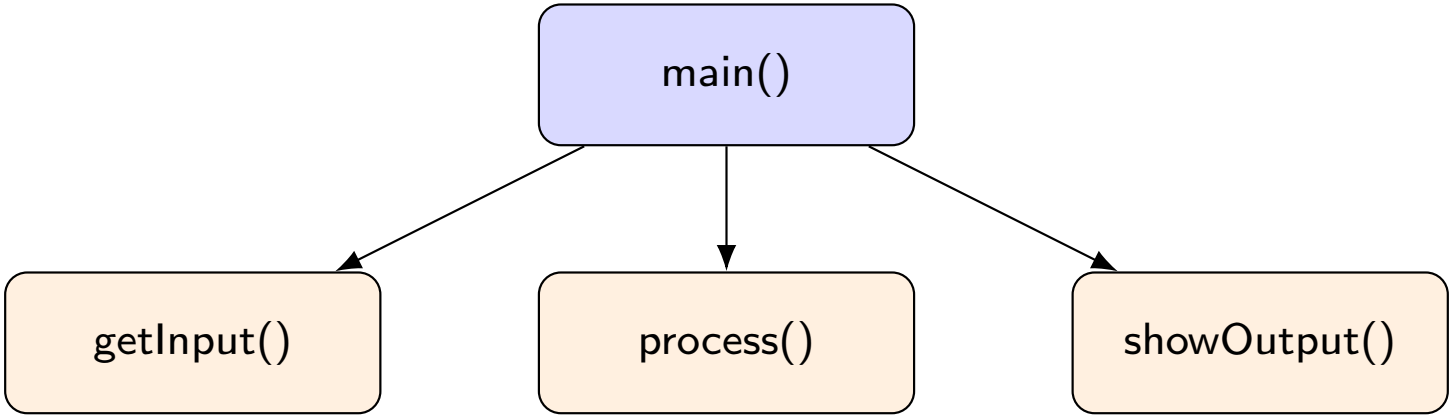


- Machine and assembly code: direct instructions to hardware, no abstraction.
- Procedural languages: introduced functions and structured control flow.
- Object-oriented languages: introduced objects that bundle data and behavior together.

Procedural Programming

- Program = sequence of instructions (Input $\rightarrow$ Processing $\rightarrow$ Output).
- Small programs: simple instruction flow is sufficient.
- Large programs: hard to manage without structure.
- Functions:
  - Break program into smaller, reusable units
  - Also called subroutines, subprograms, or procedures
  - Each function should:
    - serve a clear purpose
    - have a well-defined interface (inputs/outputs)

Procedural Programming

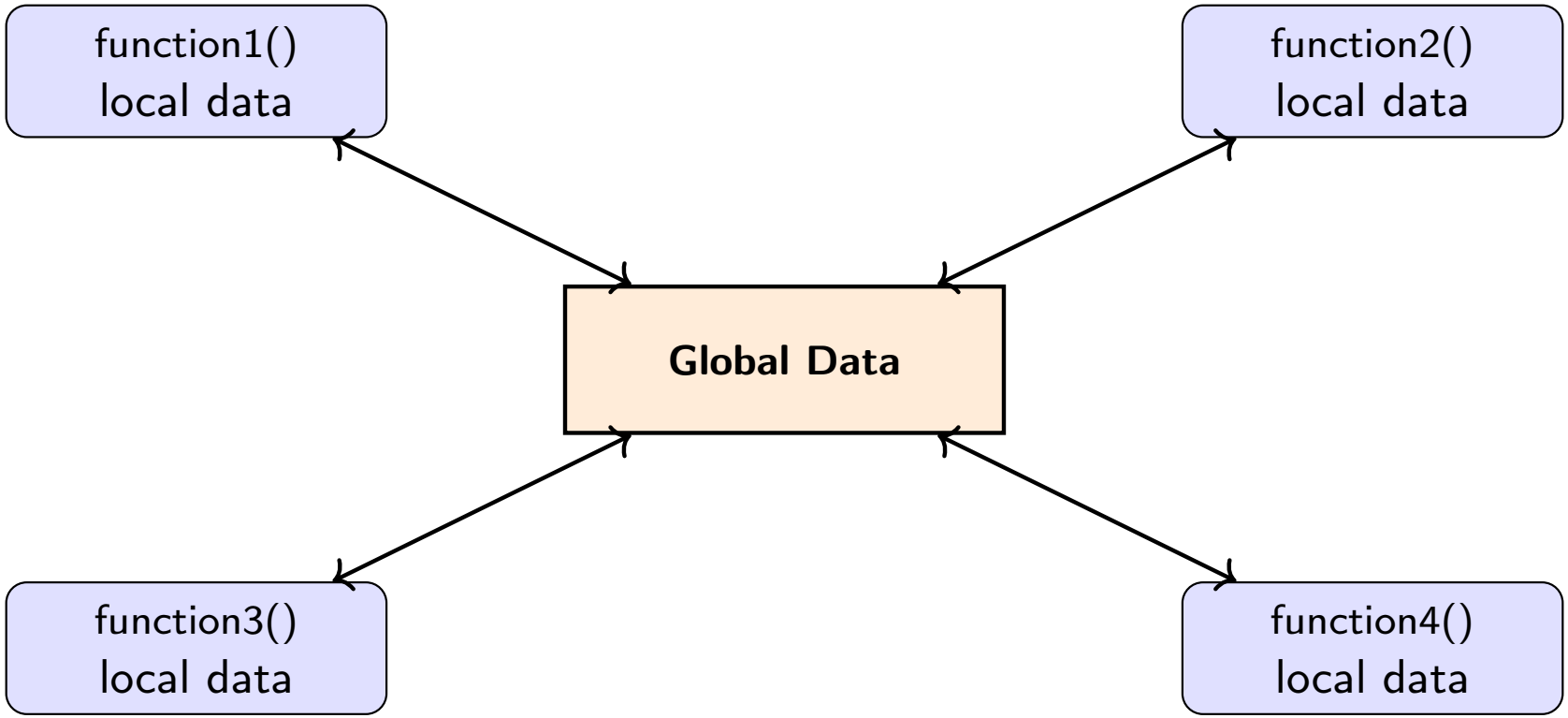


- `main()` controls the overall flow and calls each function in turn.
- Data is passed between functions as parameters and return values.
- Focus is on *what steps* the program performs, not on the data itself.

Unrestricted Access in Procedural Programming

- **Local data:** used only within one function; protected from modification by other functions.
- **Global data:** accessible by any function; useful for sharing data, but removes that protection.
- **Why this is risky:**
  - Complexity: many functions may depend on the same global data.
  - Fragility: a change to global data can require updates across every function that uses it.
- **Example:** changing a global variable from `int` to `float` means checking, and possibly fixing, every function that touches it.

Global and Local Data



- Any function can read or modify global data: there is no controlled access.
- Local data inside a function stays private to that function.

Why Global Data Is Risky

```
1  float balance;   // global data, used everywhere
2
3  void deposit(float amt) { balance += amt; }
4  void withdraw(float amt) { balance -= amt; }
5  void printBalance() { cout << balance; }
6
7  // Later, someone changes balance to an int
8  // to "save memory". Now every function above
9  // must be checked and possibly fixed by hand.
10 int balance;
```

- Three unrelated functions all depend on one global variable.
- A single, well-intentioned change (`float` to `int`) can silently break code far away from the change itself.
- As a program grows to dozens of functions and global variables, tracing these dependencies by hand becomes unmanageable.

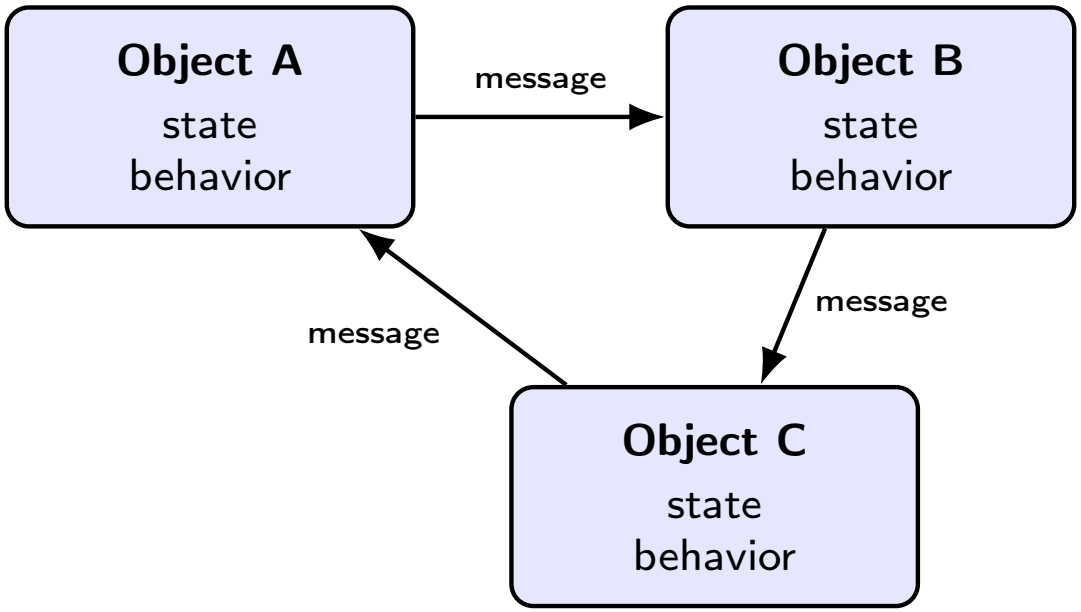
Real-World Modeling and Procedural Programming

- Procedural programming struggles to model real-world objects effectively.
- Real-world objects have:
  - **Attributes:** characteristics such as eye color (people) or horsepower (cars). Equivalent to **data** in programs.
  - **Behavior:** actions in response to stimuli, such as a person responding to a raise request or a car stopping when brakes are applied. Equivalent to **functions** in programs.
- Data and functions alone do not fully capture the complexity of objects, which integrate both attributes and behavior.

Object-Oriented Programming

- The real world consists of objects.
- Programs model real-world objects as computational entities.
- Objects have:
  - **State (attributes):** characteristics that can change.
  - **Behavior:** actions they can perform.
- Object-oriented programming shifts the focus from dividing problems into functions to dividing them into objects, giving data priority over functions in problem-solving.

Object-Oriented Paradigm



- Each object bundles its own data (state) with the functions that operate on it (behavior).
- Objects communicate by sending each other messages, that is, by calling one another's member functions.

Object-Oriented Paradigm

- A program typically consists of a number of objects.
- Data items belonging to an object are referred to as attributes or instance variables.
- Objects communicate by calling each other's member functions.
- Calling a member function of an object is often referred to as sending a message to the object.

Procedural vs Object-Oriented Programming

Procedural	**Object-Oriented**
Organized around functions	Organized around objects
Data and functions are separate	Data and functions are bundled together
Data is often global and openly accessible	Data is typically private, accessed through methods
Adding a feature can mean editing many functions	Adding a feature often means adding or extending a class
Harder to reuse across projects	Classes are easier to reuse across projects

Four Pillars of OOP

1. Abstraction
2. Encapsulation
3. Inheritance
4. Polymorphism

Abstraction

- Abstraction simplifies complex systems by *exposing only essential details and hiding unnecessary ones*.
- Reduces complexity by focusing on the critical aspects relevant to the user.
- Examples:
  - Online Shopping: When you shop online, you simply add items to your cart and make a payment. Behind the scenes, various complex processes such as inventory management, order processing, and payment verification are happening, but they are hidden from the customer.
  - Using a Smartphone: When you use a smartphone, you interact with apps to make a call, send messages, or browse the internet. However, internal processes such as network communication, data encryption, and memory management are abstracted from the user.

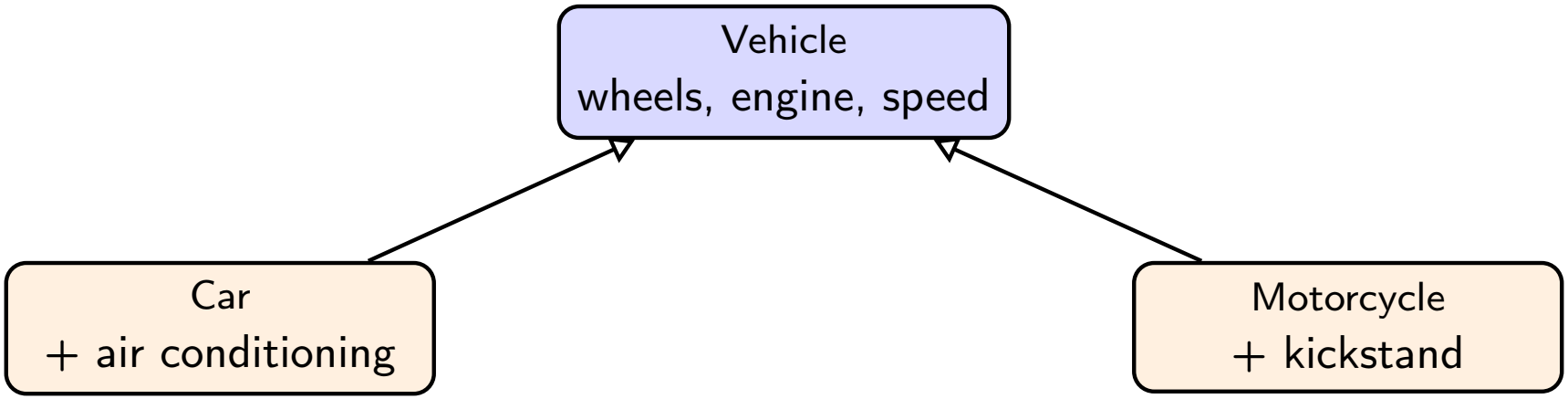
Encapsulation

- Encapsulation is the practice of *wrapping data* and *methods* that operate on that data *into a single unit*.
- Protects the data by *restricting direct access* and provides *controlled interfaces for interaction*.
- Examples:
  - Banking system: In a banking system, your account balance is private, and you can only interact with it through defined methods like deposit() or withdraw(). You can't directly modify the balance variable.
  - Smartphone: You use apps like the camera or messages, but how the phone processes these tasks (e.g., image processing or data handling) is hidden from the user. The only access you have is through the user-friendly interface.

Inheritance

- Inheritance allows a new class (child) to inherit properties and behaviors from an existing class (parent).
- Enables code reusability and creates a hierarchical relationship among classes.
- Examples:
  - Vehicles: Vehicle class may define common characteristics like wheels, engine, and speed. Car class inherits these attributes but adds unique behaviors like having air conditioning or a stereo system.
  - Electronic Devices: Device class could include shared attributes such as power source and manufacturer. Smartphone class could inherit these features and add specific properties like a touchscreen, camera, and apps.

Inheritance



- `Car` and `Motorcycle` automatically get everything `Vehicle` defines.
- Each child class only needs to add what makes it different.

Polymorphism

- Polymorphism allows objects of different types to be treated as objects of a common parent class.
- Types:
  1. **Compile-time:** same method name used with different parameters.
  2. **Run-time:** a subclass provides its own implementation of a method defined in the parent class.
- Examples:
  - **Remote control:** the same "power" button has a different effect on a TV vs. an AC.
  - **Payment system:** paying by card, PayPal, or crypto all trigger "process payment," but each works differently underneath.

Compile-Time Polymorphism: A First Look

```
1  // Compile-time polymorphism: same name, different parameters
2  int add(int a, int b) {
3      return a + b;
4  }
5
6  double add(double a, double b) {
7      return a + b;
8  }
9
10 // add(2, 3) calls the int version
11 // add(2.5, 3.5) calls the double version
```

- Two functions share the name add, but take different parameter types.
- The compiler decides which one to call based on the arguments used at the call site.
- This is called **function overloading**. We will see run-time polymorphism, using inheritance, once classes are introduced in the next lecture.

Four Pillars of OOP

1. Abstraction: what it does, not how it works.
2. Encapsulation: protect the data, expose only what's needed.
3. Inheritance: reuse features from parent to child.
4. Polymorphism: one name, many forms.

Which Pillar Is It?

For each scenario, decide which pillar of OOP it best illustrates: abstraction, encapsulation, inheritance, or polymorphism.

1. A television remote has one "volume up" button that works differently on a TV, a sound bar, or a set-top box.
2. An ATM screen only shows `Withdraw`, `Deposit`, and `Check Balance`, hiding the network calls to the bank's servers.
3. A `SavingsAccount` class and a `CurrentAccount` class both reuse the common fields and methods defined in an `Account` class.
4. A `Student` class only allows the GPA field to be changed through an `updateGPA()` method, never directly.

Why OOP?

- Lower effort, complexity, and cost of software development.
- Easier maintenance and quicker adaptation to change.
- Flexibility and reusability of code.
- Higher reliability and robustness of systems.