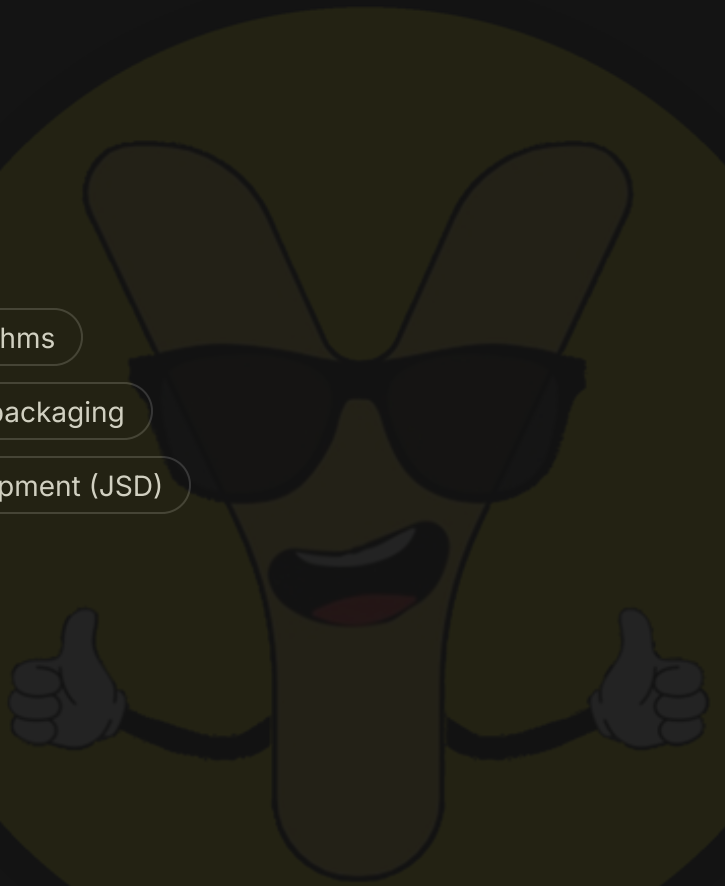
AKTU · 3RD YEAR · SEMESTER 5 · BCS054

Object Oriented System Design with C++

Unit 3 — Object Oriented Analysis and Design

- Object Oriented Analysis
- Object oriented design
- Object design
- Combining three models
- Designing algorithms
- Design optimization
- Implementation of control
- Adjustment of inheritance
- Object representation
- Physical packaging
- Documenting design considerations
- Structured analysis and structured design (SA/SD)
- Jackson Structured Development (JSD)
- Mapping object-oriented concepts using non-object-oriented language
- Translating classes into data structures



DECODED BY

Chirag Jain

Founder, Ycotes · Gold Medalist in Artificial Intelligence, IIT Madras · taught 10,000+ students

Object-Oriented Analysis (OOA)

OBJECT-ORIENTED ANALYSIS

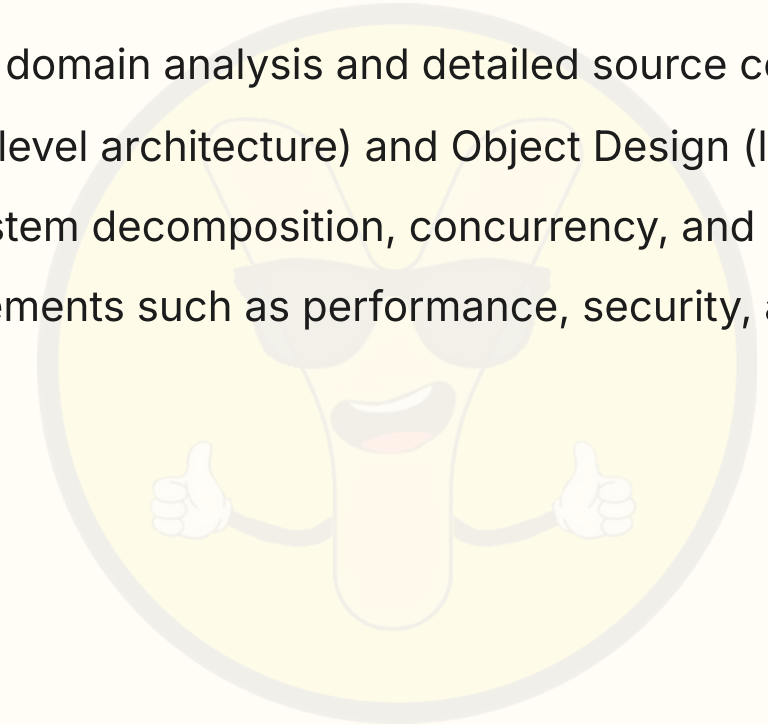
OOA is the phase of software development where system requirements are analyzed from the perspective of classes and objects identified in the problem domain.

- Focuses on understanding 'what' the system should do, not 'how' it will be built.
- Identifies real-world domain objects, their attributes, behavior, and relationships.
- Produces key domain models: Object Model, Dynamic Model, and Functional Model.
- Inputs include user requirement documents, domain knowledge, and use case specifications.

Object-Oriented Design (OOD)

OOD translates analysis models into logical software specifications ready for implementation.

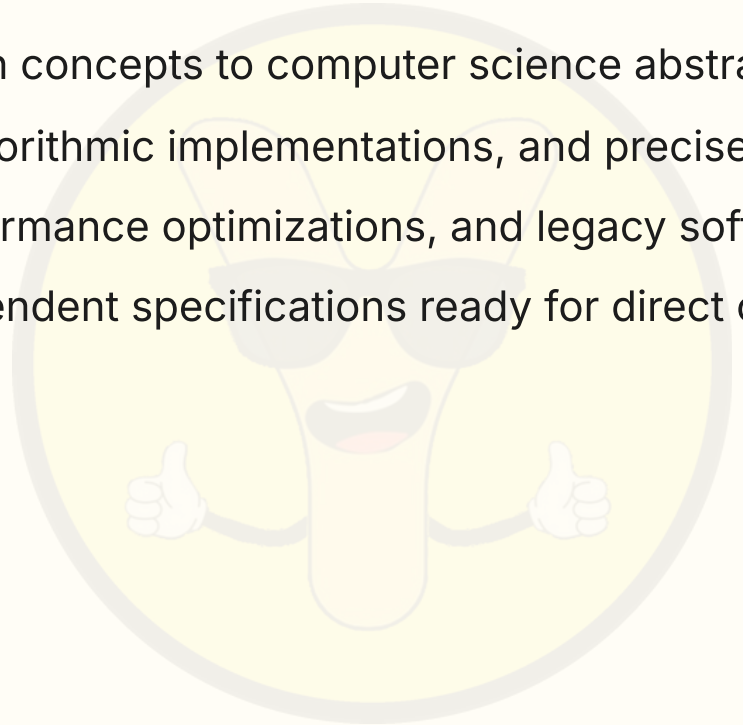
- Bridges the gap between problem domain analysis and detailed source code.
- Divided into System Design (high-level architecture) and Object Design (low-level details).
- System Design determines subsystem decomposition, concurrency, and data management.
- Focuses on non-functional requirements such as performance, security, and scalability.



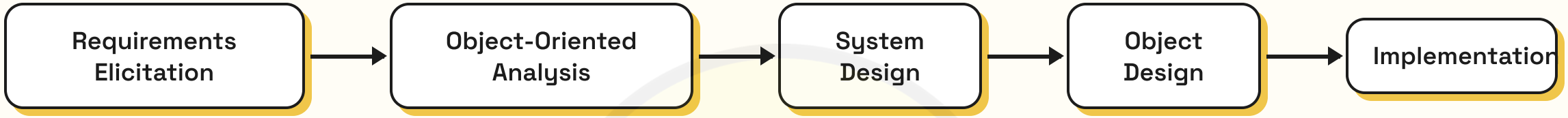
Object Design Phase

Object design refines analysis models and adds implementation-level technical details.

- Shifts focus from real-world domain concepts to computer science abstractions.
- Specifies exact data structures, algorithmic implementations, and precise method signatures.
- Incorporates design patterns, performance optimizations, and legacy software integration.
- Prepares detailed, language-independent specifications ready for direct coding.



Phases of OOAD Lifecycle

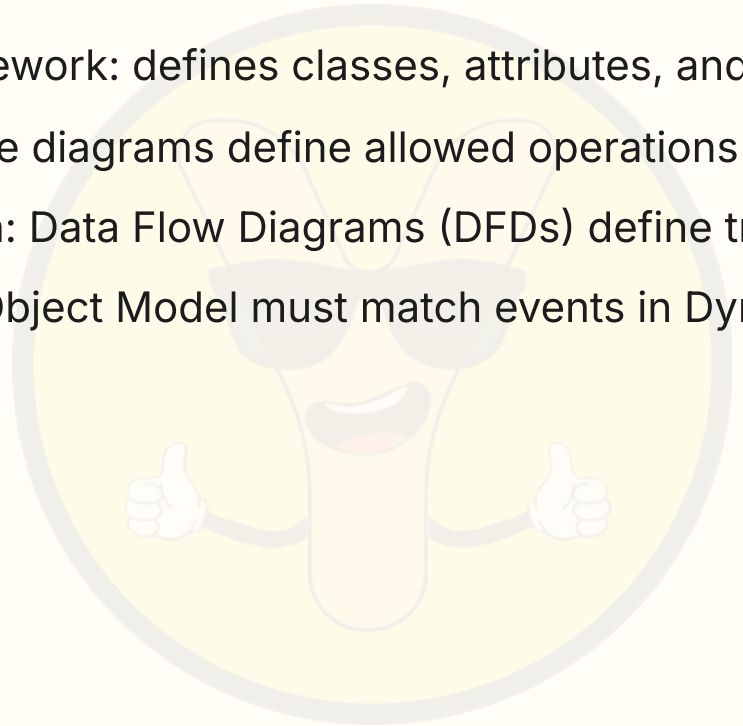


Progression from problem domain analysis to detailed code implementation.

Combining the Three OMT Models

Object Modeling Technique (OMT) integrates structural, dynamic, and functional perspectives.

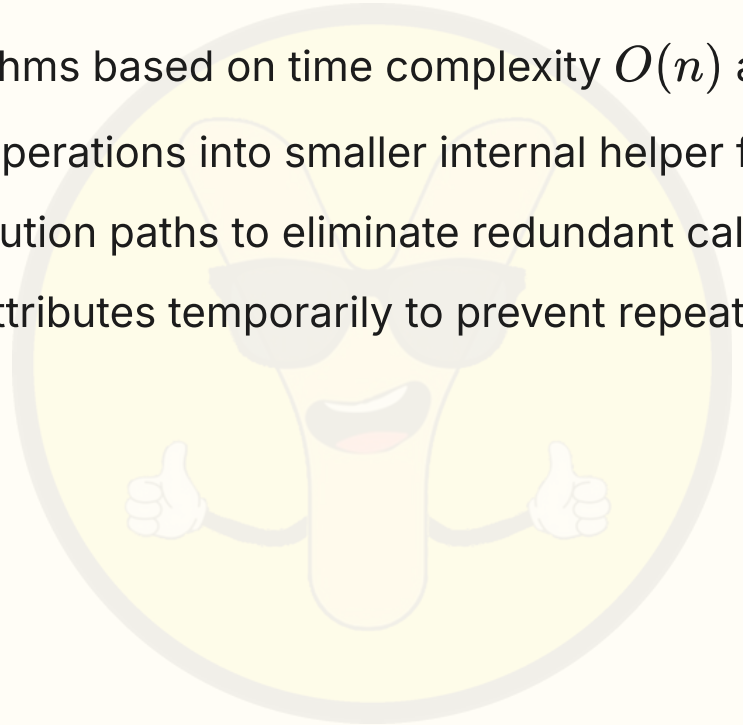
- Object Model forms the static framework: defines classes, attributes, and relationships.
- Dynamic Model adds behavior: state diagrams define allowed operations and state transitions.
- Functional Model adds computation: Data Flow Diagrams (DFDs) define transformation logic.
- Consistency check: Operations in Object Model must match events in Dynamic Model and processes in DFDs.



Designing Algorithms and Optimization

Algorithm design focuses on computational logic while optimization enhances efficiency.

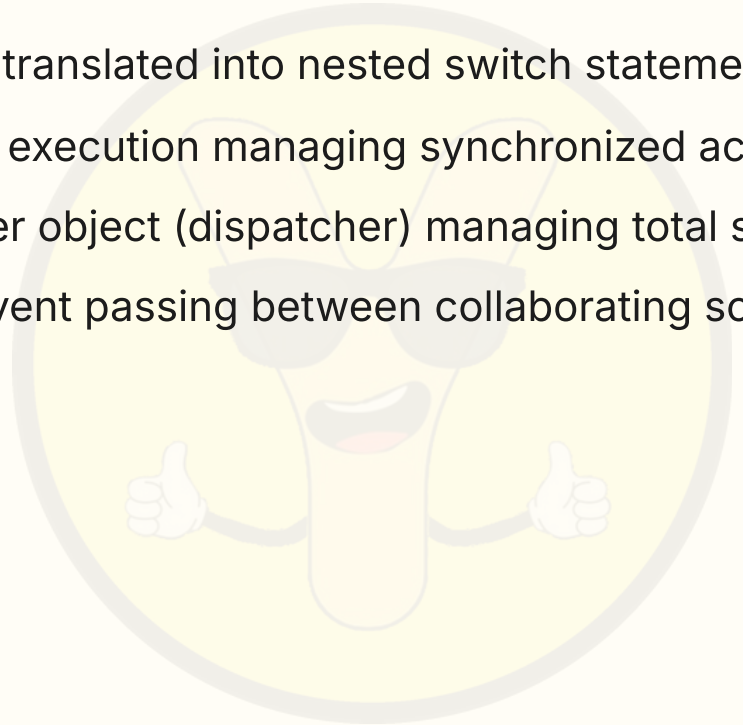
- Algorithm Selection: Choose algorithms based on time complexity $O(n)$ and memory usage.
- Refinement: Decompose complex operations into smaller internal helper functions.
- Path Optimization: Reorganize execution paths to eliminate redundant calculations.
- Execution Caching: Store derived attributes temporarily to prevent repeated dynamic evaluations.



Implementation of Control

Control strategies determine how system operations are sequenced and executed.

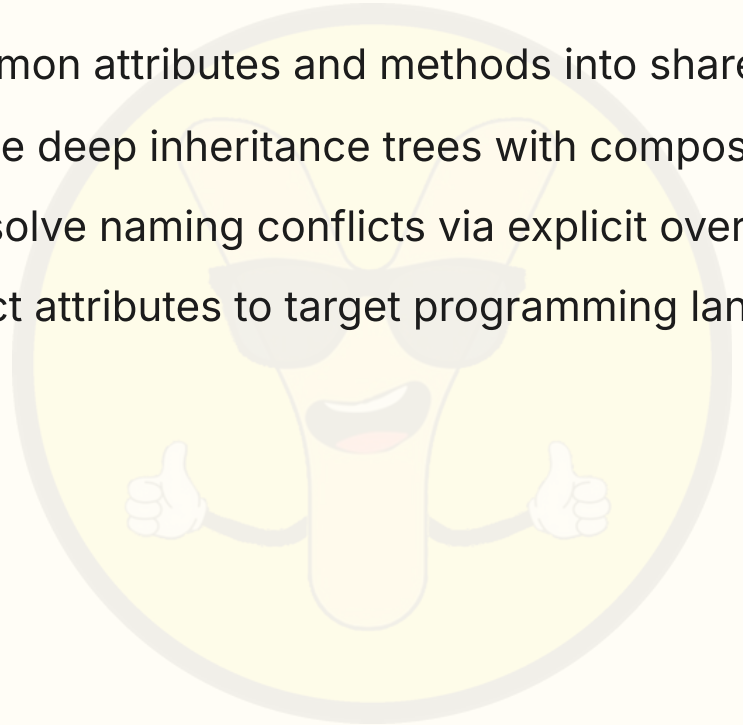
- Sequential Control: State machines translated into nested switch statements or lookup tables.
- Concurrent Control: Multi-threaded execution managing synchronized access to shared objects.
- Centralized Control: Single controller object (dispatcher) managing total system workflow.
- Distributed Control: Peer-to-peer event passing between collaborating software objects.



Adjustment of Inheritance and Representation

Fine-tuning inheritance hierarchies optimizes reusability and maintainability.

- Rearranging Classes: Abstract common attributes and methods into shared parent classes.
- Delegation over Inheritance: Replace deep inheritance trees with composition to reduce coupling.
- Multiple Inheritance Resolution: Resolve naming conflicts via explicit overriding or delegation.
- Object Representation: Map abstract attributes to target programming language primitive data types.



Physical Packaging and Documentation

Physical file organization and clear documentation ensure long-term maintainability.

- Physical Packaging: Group related classes, interfaces, and types into modules or header files.
- Information Hiding: Expose minimal public interfaces while hiding internal data implementations.
- Traceability: Document design decisions linking every design component back to analysis requirements.
- Document Contents: Includes system architecture, class dictionary, interaction diagrams, and algorithm details.

Structured Analysis and Structured Design (SA/SD)

SA/SD

SA/SD is a traditional software engineering methodology based on functional decomposition, dividing systems into data flows and process hierarchies.

- Structured Analysis (SA): Uses Data Flow Diagrams (DFDs) and Data Dictionaries to model functions.
- Structured Design (SD): Transforms DFDs into Structure Charts showing module call hierarchies.
- Separation: Data structures and functional processes are kept strictly separate.

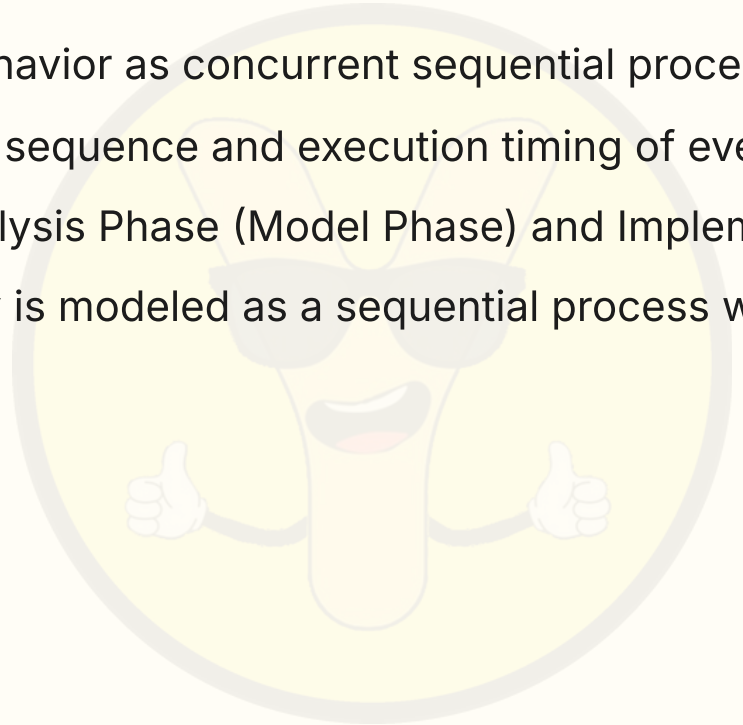
SA/SD vs Object-Oriented Analysis & Design (OOAD)

Feature	SA/SD Approach	OOAD Approach
Primary Focus	Functions and data flow processes	Objects containing data and methods
Modularity	Functional decomposition into modules	Encapsulated classes and instances
Data & Behavior	Kept separate (DFDs & Dictionaries)	Encapsulated together in objects
Reusability	Low reusability across projects	High reusability via inheritance
Change Impact	High (data schema changes affect functions)	Low (changes localized to classes)
Artifacts	DFDs, Structure Charts	UML Class, Sequence, State Diagrams

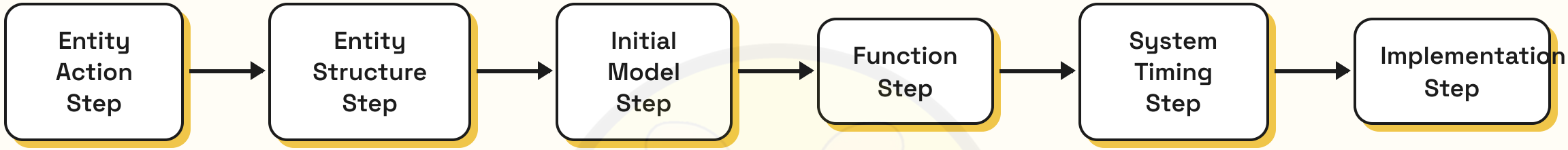
Jackson Structured Development (JSD)

JSD is a linear, process-driven software development method focused on real-world timing.

- Process-Driven: Models system behavior as concurrent sequential processes matching real-world entities.
- Timing & Ordering: Emphasizes the sequence and execution timing of events over time.
- Phased Approach: Divided into Analysis Phase (Model Phase) and Implementation Phase.
- State-Based: Each real-world entity is modeled as a sequential process with explicit states.



JSD Lifecycle Steps

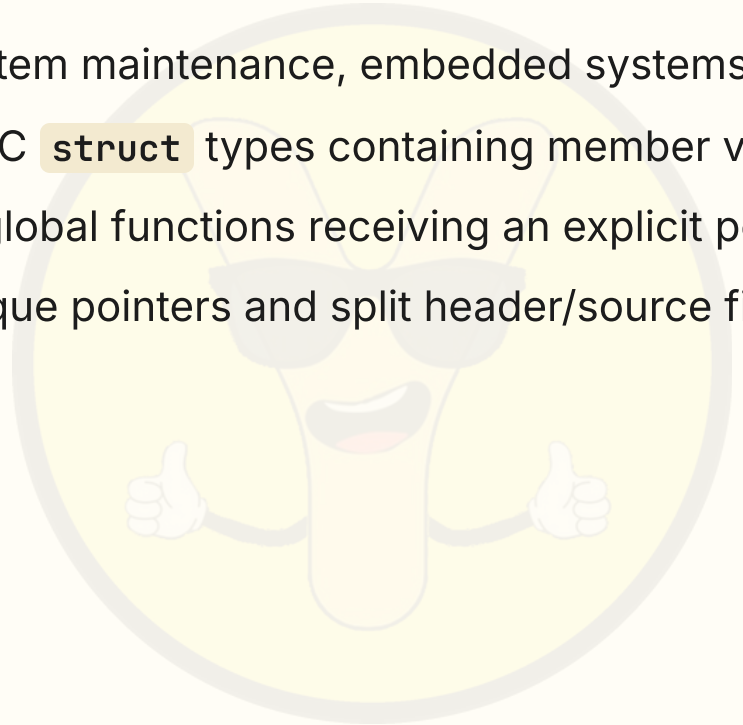


Sequential steps involved in Jackson Structured Development (JSD).

Mapping OO Concepts to Non-OO Languages

OO concepts can be manually implemented in procedural languages like C using structured constructs.

- Motivation: Required for legacy system maintenance, embedded systems, or bare-metal C applications.
- Class Mapping: Represented using C `struct` types containing member variables.
- Method Mapping: Represented as global functions receiving an explicit pointer to the struct.
- Encapsulation: Enforced using opaque pointers and split header/source file compilation units.



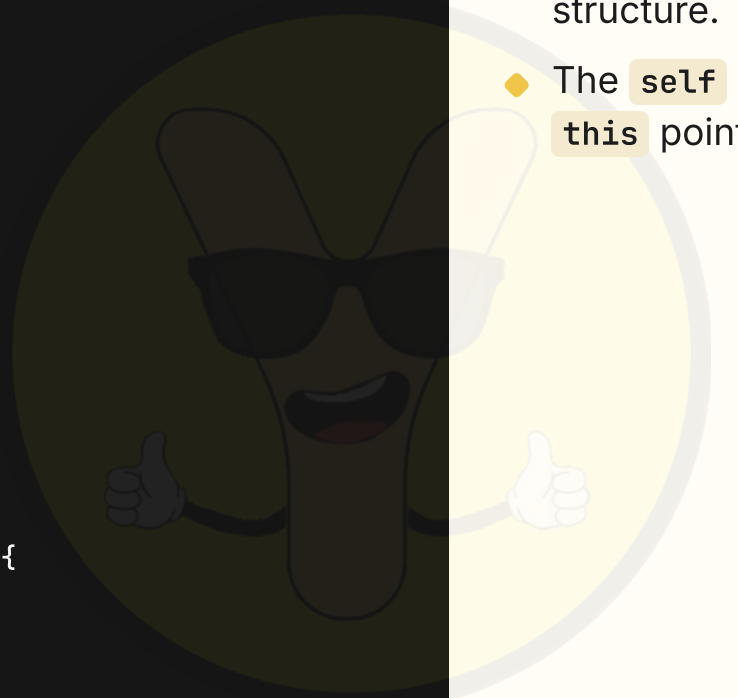
Translating Classes into Data Structures

```
/* Header File: Point.h */
typedef struct Point Point;

Point* Point_Create(int x, int y);
void Point_Destroy(Point* self);
void Point_Move(Point* self, int dx, int dy);
int Point_GetX(const Point* self);

/* Implementation File: Point.c */
struct Point {
    int x;
    int y;
};

void Point_Move(Point* self, int dx, int dy) {
    if (self) {
        self->x += dx;
        self->y += dy;
    }
}
```



- Class instance fields are declared inside a C structure.
- The `self` pointer explicitly replaces the implicit C++ `this` pointer.

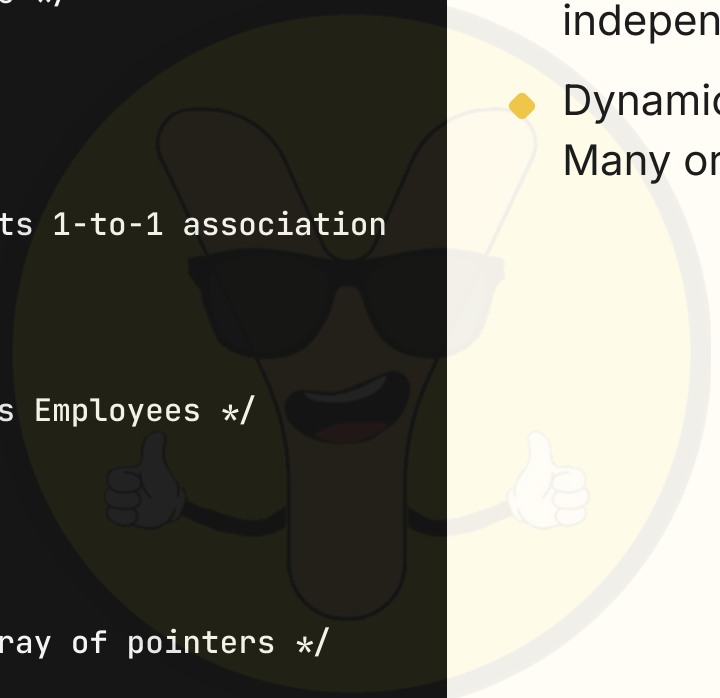
Implementing Associations in Non-OO Languages

```
/* 1-to-1 Association: Car has an Engine */
typedef struct Engine Engine;

typedef struct Car {
    char model[50];
    Engine* engine; /* Pointer represents 1-to-1 association
*/
} Car;

/* 1-to-Many Association: Department has Employees */
typedef struct Employee Employee;

typedef struct Department {
    char name[50];
    Employee** employees; /* Dynamic array of pointers */
    int count;
} Department;
```



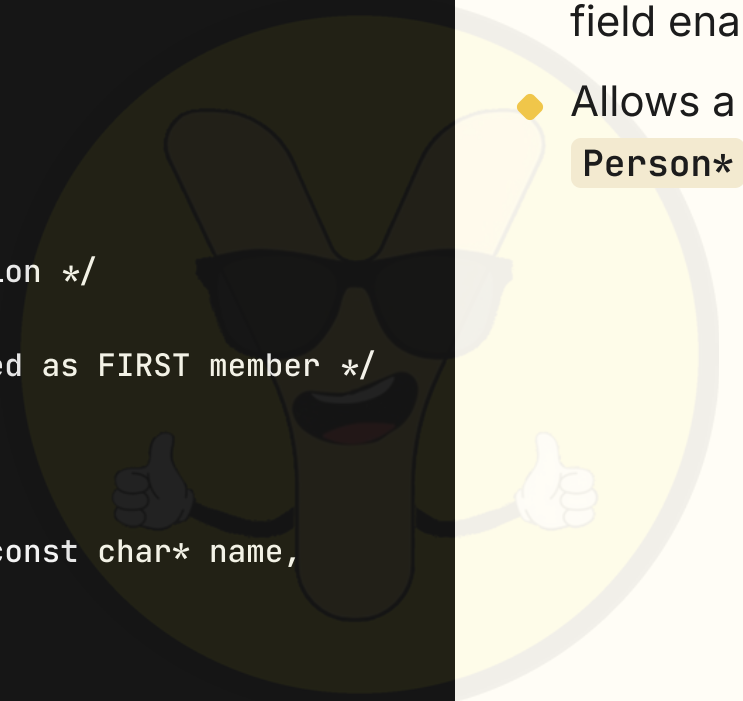
- Pointers simulate 1-to-1 associations between independent structures.
- Dynamic arrays of pointers implement 1-to-Many or Many-to-Many relationships.

Implementing Inheritance in Procedural C

```c
/* Base "Class" Struct */
typedef struct {
    int id;
    char name[50];
} Person;

/* Derived "Class" Struct via Composition */
typedef struct {
    Person base; /* Base struct embedded as FIRST member */
    float gpa;
} Student;

void Student_Init(Student* s, int id, const char* name,
float gpa) {
    s->base.id = id;
    strncpy(s->base.name, name, 49);
    s->gpa = gpa;
}
```

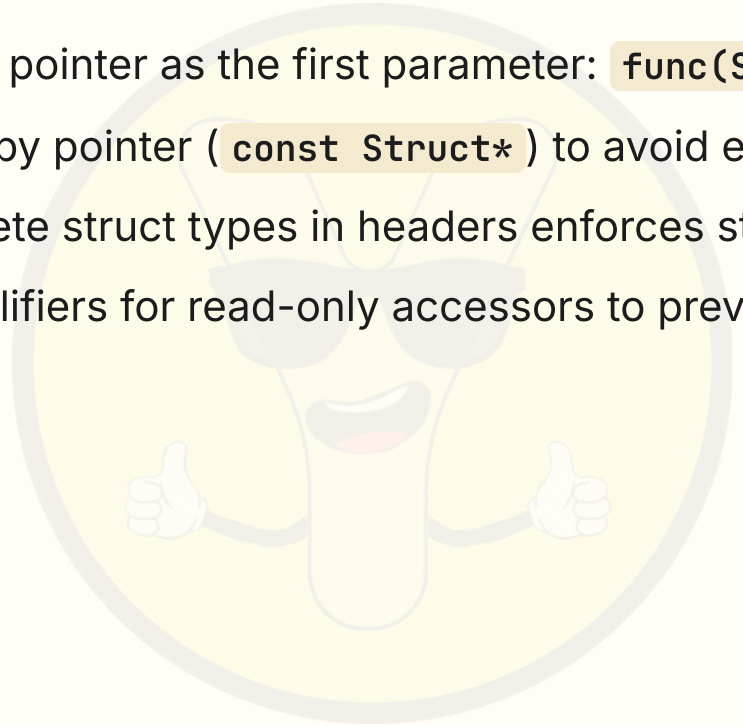


- Embedding base struct as the first memory field enables safe pointer casting.
- Allows a `Student*` pointer to be cast to a `Person*` pointer for polymorphic calls.

Encapsulation & Passing Method Arguments

Managing instance state and function parameters in procedural object mappings.

- Explicit 'this' Pointer: Pass instance pointer as the first parameter: `func(Self* self, args)`.
- Pass by Pointer: Pass large structs by pointer (`const Struct*`) to avoid expensive stack copies.
- Opaque Structs: Declaring incomplete struct types in headers enforces strict data encapsulation.
- Const Correctness: Use `const` qualifiers for read-only accessors to prevent accidental state modification.



Worked Example: Translating Class to C Struct

Translate a C++ Rectangle class with width, height attributes and a getArea() method into procedural C code.

1. Step 1: Define C struct for attributes: typedef struct { double width; double height; } Rectangle;
2. Step 2: Write initializer function: void Rectangle_Init(Rectangle* r, double w, double h) { r→width = w; r→height = h; }
3. Step 3: Define method taking explicit pointer: double Rectangle_GetArea(const Rectangle* r) { return r→width * r→height; }
4. Step 4: Execute in main: Declare Rectangle r; Rectangle_Init(&r, 5.0, 4.0); double area = Rectangle_GetArea(&r);

The implicit C++ 'this→width' maps directly to explicit pointer dereferencing 'r→width' in C.

Exam Corner: Likely AKTU Questions

- **Q1** Explain Object-Oriented Analysis (OOA) and Object-Oriented Design (OOD). Differentiate between SA/SD and OOAD.
- **Q2** Describe the process of combining Object, Dynamic, and Functional models in OMT with neat diagrams.
- **Q3** What are the major steps involved in Jackson Structured Development (JSD)? Explain with a flowchart.
- **Q4** How are OO concepts like Inheritance, Association, and Encapsulation implemented using non-OO languages like C?
- **Q5** Write short notes on: (a) Optimization of Design, (b) Physical Packaging, (c) Implementation of Control.
- **Q6** Translate a C++ class hierarchy containing a base class and a derived class into equivalent C structures and functions.

Unit 3 done. You are exam-ready.

Revise the Exam corner once more, then open Unit 4 on theycotes.com. Every AKTU subject, unit by unit, free. Notes by Chirag Jain, Ycotes.

