# Welcome

# Object Oriented Programming (UTA018)

**(Session: 2026-2027 ODD)**

Instructor: Dr. Suresh Raikwar

**Department of Computer Science and Engineering,**

**Thapar Institute of Engineering and Technology, Patiala, Punjab, India**

**Important Instruction(s):**

- This material is intended for guidance and conceptual understanding. It does not guarantee that examination questions will be drawn directly from it.
- Merely studying this material is not sufficient for achieving good performance.
- Sessional Tests, MST, and EST will assess your understanding of Object-Oriented Programming concepts through **logical reasoning, analytical thinking, and problem-solving**.
- Students are strongly encouraged to attend regular classes and practice programming problems from multiple sources to develop a deeper conceptual understanding and improve problem-solving skills.


दृश्य एवं संकेत सूचना प्रसंस्करण समूह
Visual Signal and Information Processing Group
TIET



# Week 1

# Introduction

- Models real-world entities using objects, classes, and their interactions.
- Built on four core principles: Encapsulation, Abstraction, Inheritance, and Polymorphism.
- Promotes reusable and modular code, reducing development time.
- Improves maintainability and scalability of software systems.
- Widely used in software development, mobile apps, games, AI, cloud, and enterprise applications.
- Develops problem-solving and software design skills essential for industry.
- Hands-on implementation using C++ with real-world programming exercises.

**Course Objective** : To become familiar with object oriented programming concepts and be able to apply these concepts in solving diverse range of applications.
**Lectures**: 03 contact hours per week,
**Lab**: 02 contact hours per week
**Course Credit**: 04

# Introduction (*contd…*)

**Syllabus:** Divided in five units as:

**Unit-1 (Objects and Classes):** Structure in C and C++, Class specification, Objects, Namespaces, Overview of pillars of OOPS (Data Encapsulation, Data Abstraction, Inheritance, Polymorphism), Inline functions, Passing objects as arguments, Returning object from a function, Array of objects, Static keyword with data member, member function and object, Friend function, and Friend classes, Pointer to objects, this pointer, Dynamic Initialization, Dynamic memory allocation.

**Unit-2 (Constructor and Destructor):** Constructors and its types, Constructor Overloading, Constructors in array of objects, Constructors with default arguments, Dynamic Constructor, Destructor, _const_ keyword with data member, member function and object. Case Study on Optimizing Power Consumption in Smart City Energy Grids.

**Unit-3 (Inheritance):** Introduction to Inheritance, Forms of Inheritance (Single, Multiple, Multilevel, Hierarchical and Hybrid) with various modes (Public, Private and Protected), Inheritance with Constructor and Destructor, Benefits and Limitations of Inheritance.

**Unit-4 (Polymorphism):** Classification of Polymorphism (Compile-time and Run-time), **Compile Time-**Function Overloading, Operator Overloading (Unary operator and Binary operator with member function and friend function), Data Conversion (Basic to user-defined, user- defined to basic, one user-defined to another user-defined). **Run-time-** Pointers to derived class object, Overriding member function, Virtual functions, pure virtual functions, Abstract class. Case Study on Developing an Eco-Friendly Smart Transportation System.

**Unit-5 (Exception Handling, Templates and Standard Template Library):** Exception handling mechanism, Usage of template, Function templates, Overloading of Function templates, Class templates, Introduction to Standard Template Library and its components. Algorithms, Containers (Array, Vector, Stack, List and Queue) and Iterators.

# Introduction (*contd…*)

## Course Learning Outcomes (CLO)

**CLO-1**: To recall the knowledge of structure and its variables to comprehend the concept of classes, objects, constructors and destructors for implementing the object oriented paradigms.

**CLO-2**: To apply and analyze the inheritance on real life case studies via coding competences.

**CLO-3**: To design and develop code snippets for polymorphism to proclaim coding potential; and management of run-time exceptions.

**CLO-4**: To assess and interpret the knowledge of templates to appraise the standard template libraries.

# Introduction (*contd…*)

**Books to be referred**

**Text Books (TB)**

TB1. C++:The Complete Reference , Schildt H., Tata McGraw Hill, 4th ed, 2003

TB2. C++Primer, Lippman B.S., Lajoie J., and MooE.B., Addison-Wesley Professional, 5th ed, 2013

**Reference Books (RB)**

RB1. Object-Oriented Programming in C++, Lafore R., Pearson Education, 4th ed, 2002

RB2. Object Oriented Programming with C++, E Balagurusamy, 8th ed,2017

**[Strongly Recommended]**

RB3. The C++programming language, Stroustrup B., Pearson Education India, 4th ed, 2013 **[Must read for competition]**

# Introduction (*contd…*)

## Importance of the Object Oriented Programming for the career

| Reason | Benefits |
|---|---|
| **Industry Standard** | Most modern software is developed using OOP languages such as C++, Java, C#, Python, etc. |
| **Foundation for Multiple Languages** | Once you understand OOP, learning a new OOP language becomes much easier. |
| **Problem-Solving Approach** | Learn to model real-world systems using objects, classes, and relationships. |
| **Code Reusability** | Write code once and reuse it through inheritance and composition, reducing development effort. |
| **Easy Maintenance** | Modular programs are easier to debug, update, and extend. |
| **Scalable Software Development** | OOP enables teams to build large and complex applications efficiently. |
| **Career Readiness** | OOP is a core skill tested in coding interviews, internships, and software engineering roles. |
| **Foundation for Advanced Technologies** | Essential for AI applications, game development, mobile apps, cloud computing, enterprise software, IoT and in many more applications. |

# Tentative Evaluation Scheme

| SN | Component | Marks | Syllabus | Date/Duration |
|---|---|---|---|---|
| **1** | Sessional | 20 | Syllabus (Unit-I, Unit-II and Unit-III) in class and syllabus covered in lab. | Week-4 to week-8 during lab hours |
| **2** | MST | 30 | Unit-I, Unit-II and Unit-III | As per Institute Examination Schedule |
| **3** | Sessional | 10 | Complete Course Syllabus (Unit-I to Unit-V) | Week-12 to week-16 during lab hours |
| **4** | EST | 40 | Complete Course Syllabus (Unit-I to Unit-V) | As per Institute Examination Schedule |
| **Total =** | | 100 | | |

# Note:

- The detailed evaluation scheme for the sessional components will be shared shortly.
- **100% attendance is mandatory for all classes and lab sessions.**
- A relaxation of up to 25% may be considered only in genuine cases with prior approval from the Dean of Academic Affairs (DOAA), as per institute norms.
- **There will be no make-up for any sessional component**, including quizzes, assignments, laboratory evaluations, or other assessments.
- You are encouraged to attempt the questions in the prescribed order during both the **MST** and **EST**. **1 mark** is reserved for following the prescribed sequence of questions.

# Approach to score

- Attend the classes regularly.
- Submit assignment before deadline.
- Participate in surprise coding contest launched regularly during the course.
- Explore problems on web world, read books and solve problems as much as possible.
- Improve rank on Code-Chef platform.
- Attend labs regularly and solve the problem provided during labs.

# Data Collection for codechef

**Link:** https://docs.google.com/forms/d/e/1FAIpQLScnu4__5nlFTQ7R-mSvXu_YzT989pvw_TpezUJkPmDq1h2CtA/viewform

# How to write exam?

Think more write less

# Complete details on the answer sheet

**Name**: Akash Kabir **Roll No.** 101987654 Group: **I3** (not just I or COE)

**Course No:** UTA018 **Course Name:** Object Oriented Programming

**Instructor**: *Lecture teacher* **Date**: *Exam date*

# Must follow the instructions of Q paper

- **NOTE:** Solve only 5 out of 6 questions in order, new problem on new page, if not sure, then leave appropriate pages for the leftovers. Plan on the last page before writing to avoid cutting. No partial credit and no rechecking for the messy exams. (also there are instructions in each question)

# Brief answer

**Q: Fill in the blanks in this code**

```
_____<stdio.h>  // to link this header file
int main(){
_____(“Hello world”);  // to display the message
return 0;
}
```

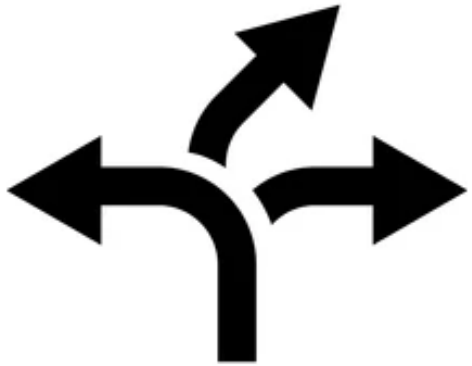
- Answer sheet

ANS:

(i) #include

(ii) printf

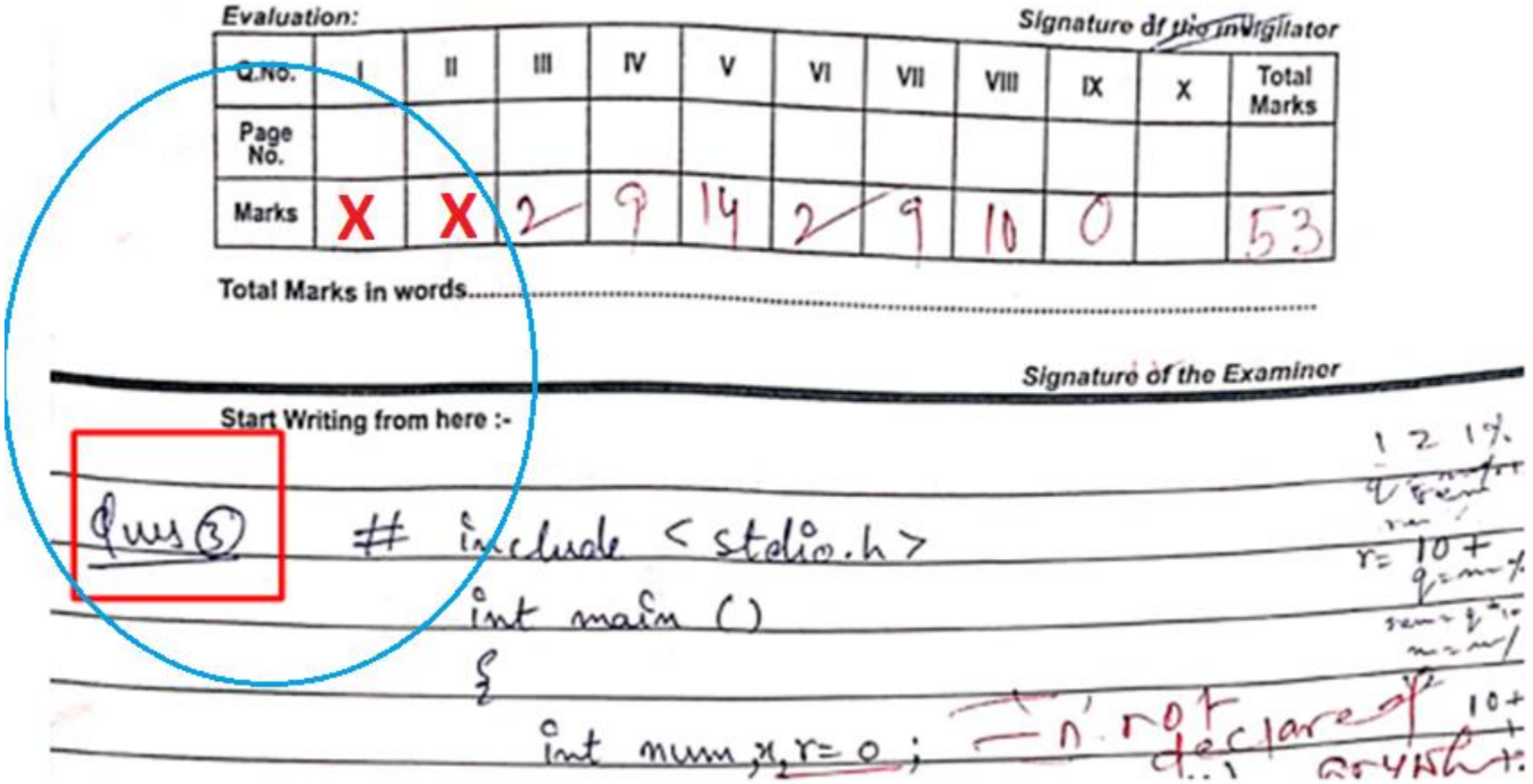
# Thinking is more important than writing




An engineer always find 2-3 alternatives, then chooses the best one.

# Plan on the last page before start writing. Why rush, when you have 2-3 hours?

# Out of order or mess will not be re-evaluated **(also no partial credit)**



Evaluation: Signature of the Invigilator

| Q.No. | I | II | III | IV | V | VI | VII | VIII | IX | X | Total Marks |
|---|---|---|---|---|---|---|---|---|---|---|---|
| Page No. | | | | | | | | | | | |
| Marks | X | X | 2 | 9 | 14 | 2 | 9 | 10 | 0 | | 53 |

Total Marks in words.....

Signature of the Examiner

Start Writing from here :-

Ques 3.

```
# include <stdio.h>
int main ()
{
int num, x, r=0;
```

n not declared

# Don’t know Q1 or Q2 and want to start Q3?

- Leave appropriate number of pages with your judgement

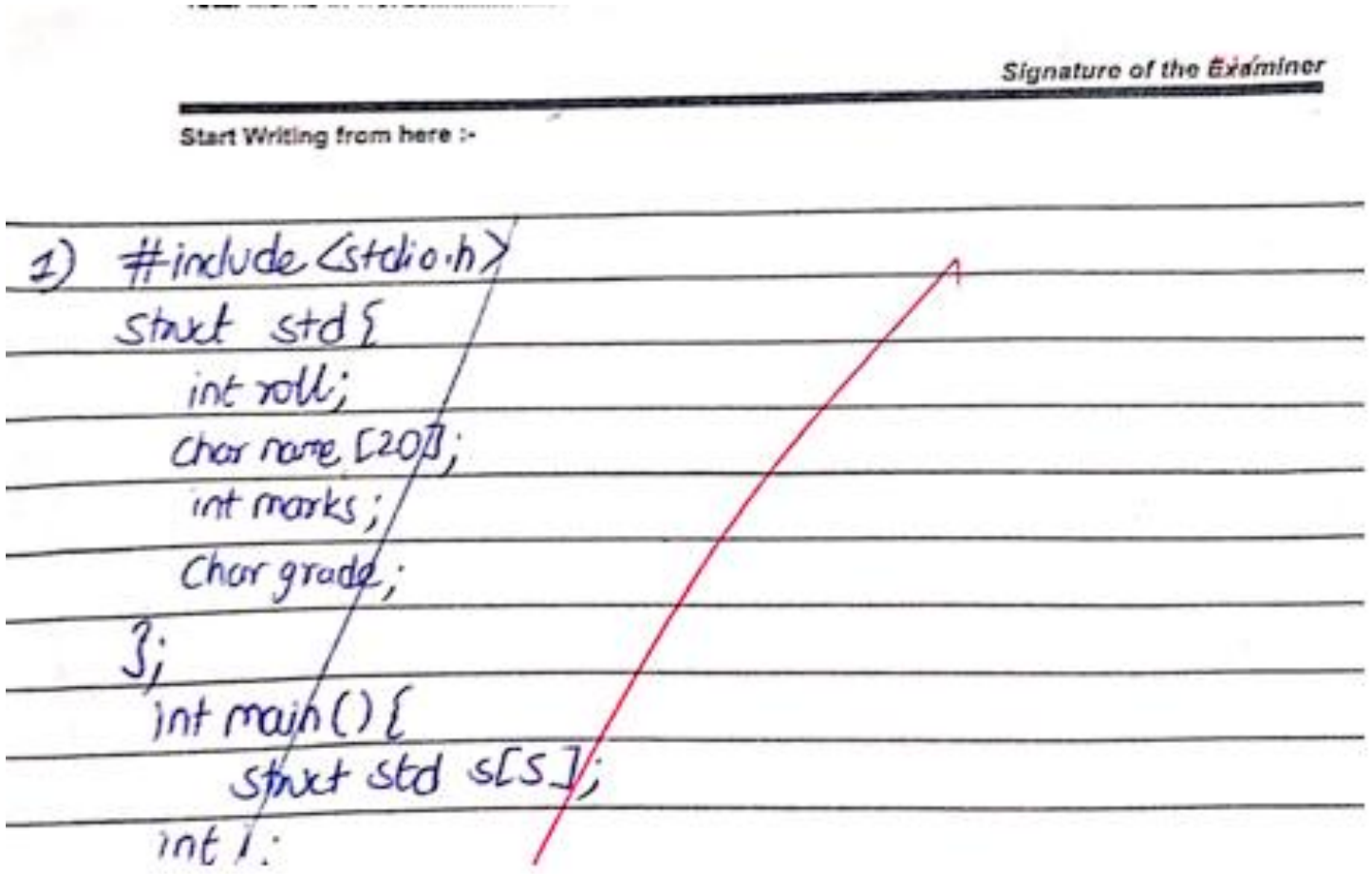
# New page for new problem

# Avoid asking for extra sheet

- Smart people are always to the point.

# Do not leave the first page blank

# Cutting on the first page is poor impression

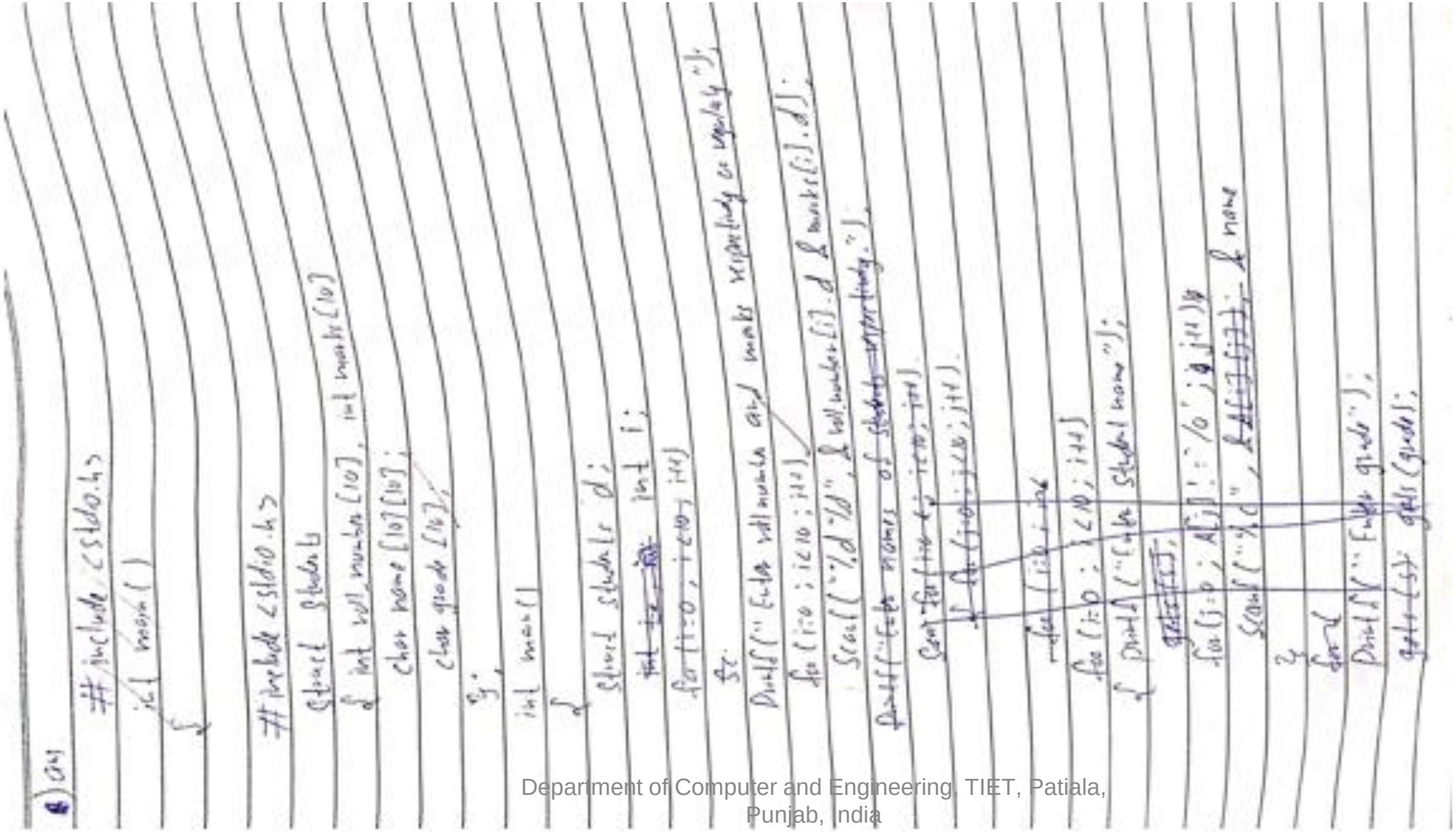


Signature of the Examiner

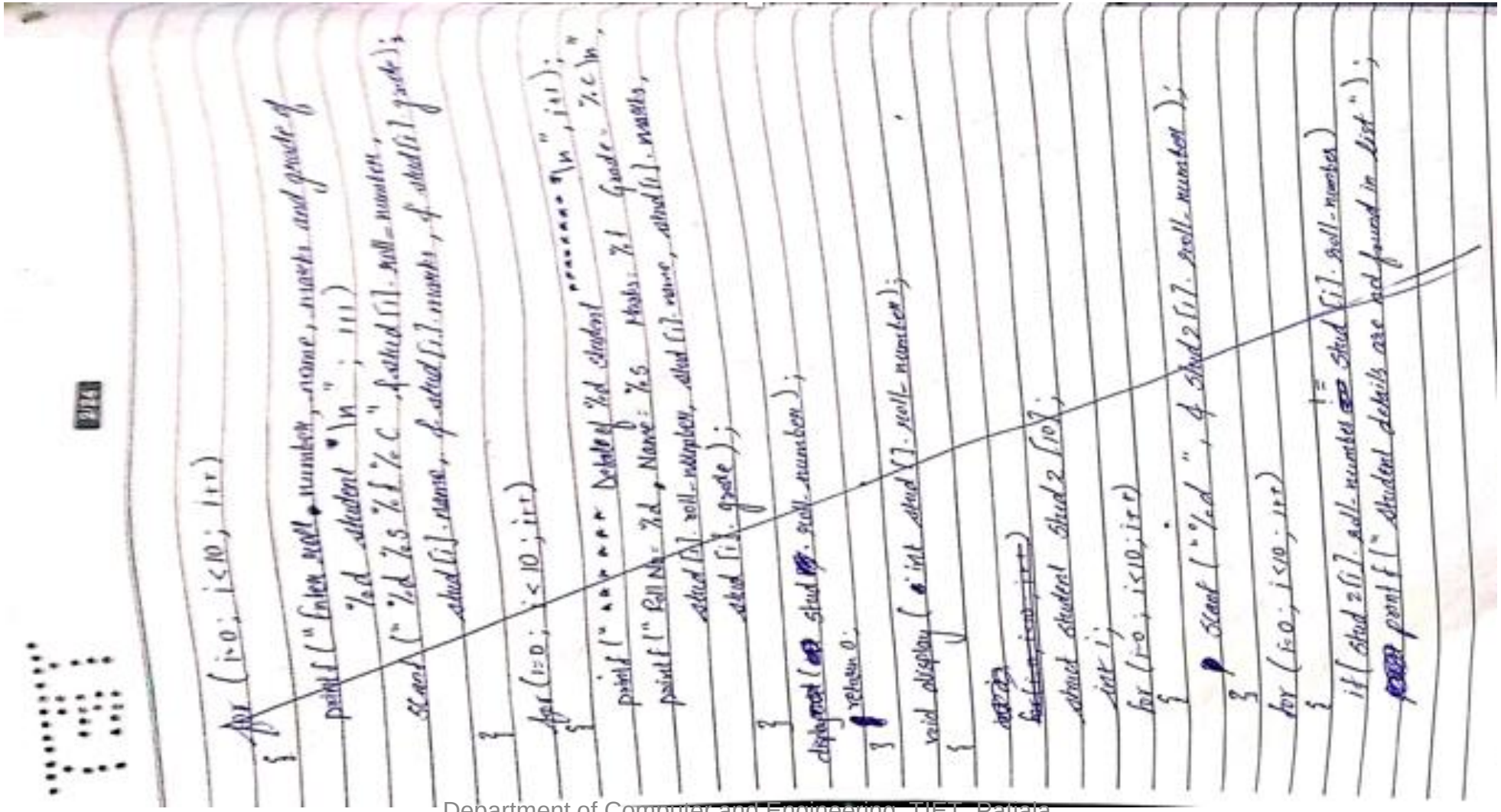
Start Writing from here :-

```
1) #include<stdio.h>
   struct std{
     int roll;
     char name[20];
     int marks;
     char grade;
   };
   int main(){
     struct std s[5];
   int i;
```

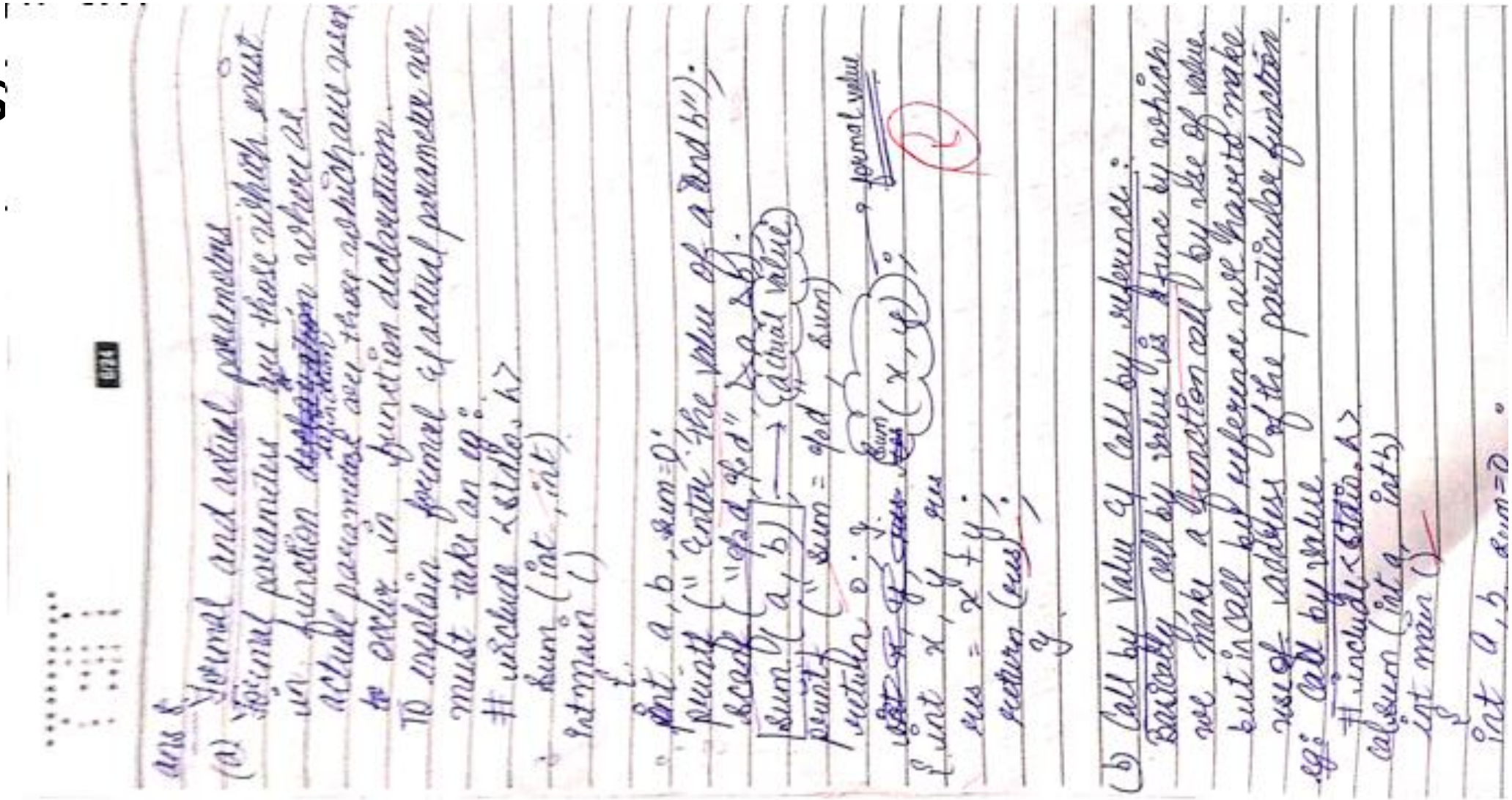
# No revaluation for such writeups



# Be conscious while writing



# Maggie noodles



# Manual vs Programming

```
s3[0] = s1[0];
s3[1] = s1[1];
s3[2] = s1[2];
s3[3] = s1[3];
s3[4] = s1[4];
s3[5] = s1[5];
s3[6] = s1[6];
s3[7] = s1[7];
s3[8] = s1[8];
```

versus

```
for(i=0;i<8;i++)
s3[i] = s1[i];
```

# Checking + rechecking of 2,000 exams *efficiently* without order, poor handwriting and cutting is impossible.

# Engineers make ***diagrams*** and ***tables*** instead of stories

| | PRODUCT A | PRODUCT B | PRODUCT C |
|---|---|---|---|
| List your feature here | | ✓ | ✓ |
| List your feature here | ✓ | | ✓ |
| List your feature here | | | ✓ |
| List your feature here | | | ✓ |
| List your feature here | | ✓ | ✓ |
| List your feature here | | ✓ | ✓ |
| List your feature here | ✓ | | ✓ |

# Mature writing

- Writing *extra* is silly and involves more mistakes.
- Box the answer

# Grace

- Write gracefully like a university student

# QAs

What is the meaning of PTO? Is it really smart?

Putting God symbols on the exam.

Who invented the theory of writing more than required gets you more points?

What does a neat exam says to a reader?

What is the advantage of instructions in the Q paper?

Should you use pencil or pen for the exam?

# Expected Outcome

O-1: Learning of Object Oriented Programming Concepts. **[Mandatory, will be evaluated]**
O-2: Improvement in Problem Solving Skills. **[Mandatory, will be evaluated]**
O-3: Exposure to Programming Problems. **[Mandatory, will be evaluated]**
O-4: Achievement of Course Objectives. **[by product]**
O-5: CodeChef Certification. **[Optional, highly recommended]**
O-6: Development of Programming Portfolio. **[Optional, highly recommended]**
O-7: Objective Resume Building. **[Optional, but highly recommended]**

## Expected benefits to you

- Strong understanding of Object Oriented Programming.
- Improved programming and problem-solving skills.
- Exposure to competitive programming.
- A measurable programming profile.
- A stronger resume supported by objective achievements.
- A programming portfolio suitable for internships and placements.
- Preparation for advanced learning, research, product development, and entrepreneurship.

# Communication Policy

To ensure that all students receive consistent information and timely responses, **students are requested not to contact individual faculty members through phone calls, WhatsApp messages, emails, or personal messages for course-related queries.**

All academic and administrative queries must be posted on the **Discussion Forum**. The Course Coordinator will regularly monitor the forum, respond to queries, and coordinate with the concerned faculty members whenever required. This ensures that responses are visible to all students, avoids repetition of similar queries, and maintains transparency.

In case of an **urgent matter** that cannot wait for a response on the Discussion Forum, students may either:

- Meet the concerned faculty member during their **official visiting hours**, or
- Contact the **Course Coordinator** for further assistance.

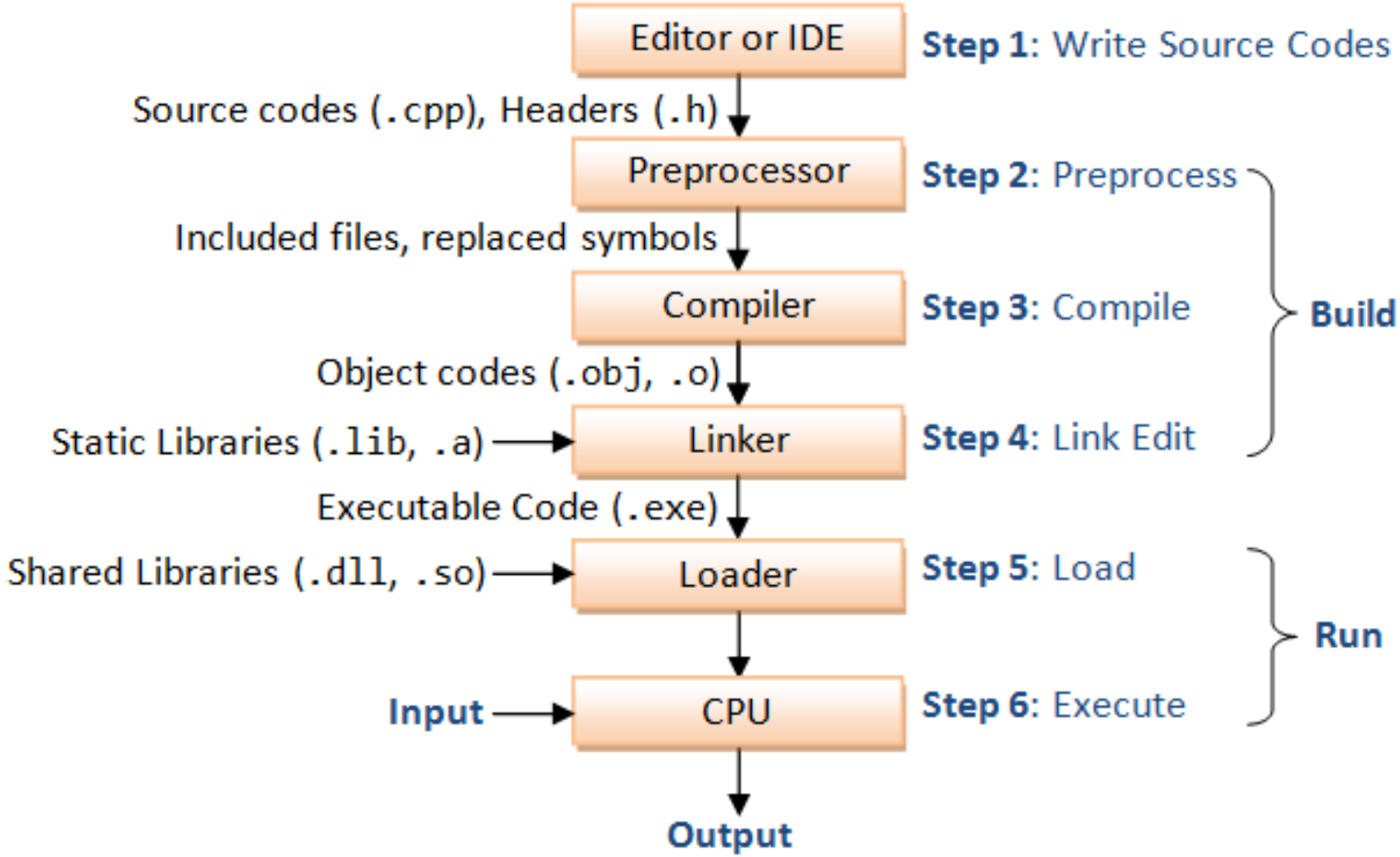
☺ **A small request:** Please let the **Discussion Forum** do the talking. It keeps the faculty happy, the students happy, and everyone's phone battery happy too! 😊

# Unit-1

Structure in C and C++, Class specification, Objects, Namespaces, Overview of pillars of OOPS (Data Encapsulation, Data Abstraction, Inheritance, Polymorphism), Inline functions, Passing objects as arguments, Returning object from a function, Array of objects, Static keyword with data member, member function and object, Friend function, and Friend classes, Pointer to objects, this pointer, Dynamic Initialization, Dynamic memory allocation.

# How a Program is Executed?

# Flowchart


Editor or IDE
Step 1: Write Source Codes
Source codes (.cpp), Headers (.h)
Preprocessor
Step 2: Preprocess
Included files, replaced symbols
Compiler
Step 3: Compile
Build
Object codes (.obj, .o)
Static Libraries (.lib, .a)
Linker
Step 4: Link Edit
Executable Code (.exe)
Shared Libraries (.dll, .so)
Loader
Step 5: Load
Run
Input
CPU
Step 6: Execute
Output

# 1. Editor

- C++ program is written in an Editor (DevC++, TurboC++, notepad, notepad++, etc.).
- Saved as a file with extension  .cpp

# 2. Preprocessor

- Preprocessing performs (usually simple) operations on the source file(s) prior to compilation.

- **Typical preprocessing operations include:**

  (a) **Expanding macros** (shorthand notations for longer constructs). For example, in C,

  #define abc(x,y) (3*x+y*(2+x))

  In program n = abc(a,b) becomes

  n = (3*a+b*(2+a))

- (b) **Inserting named files**. For example, in C++,

  # include <iostream> is replaced by the contents of the file iostream.h

**Overall:** The C++ preprocessor expands the source code before compilation by handling header files, macros, conditional compilation, and other # directives.

# 3. Tasks of the Compiler

- **Lexical Analysis**: Converts source code into tokens.
- **Syntax Analysis**: Checks whether the program follows the grammar and syntax rules.
- **Semantic Analysis**: Checks meaning, type compatibility, scope, declarations, etc.
- **Symbol Table Management**: Records variables, functions, data types, scope, and memory-related information.
- **Type Checking**: Verifies compatibility of data types in expressions and assignments.
- **Scope and Declaration Checking**: Ensures variables/functions are declared and used within valid scopes.
- **Intermediate Code Generation**: Converts the program into an intermediate representation.
- **Code Optimization**: Improves code for execution speed and/or memory efficiency.
- **Memory/Storage Allocation Information**: Determines storage requirements and offsets for variables and data structures. Do not allocate memory.
- **Target Code Generation**: Generates assembly/machine-level object code.
- **Object File Generation**: Produces the object file containing machine code, symbols, and relocation information.
- **Relocation Information Generation**: Marks addresses that need to be adjusted by the linker.
- **External Symbol Resolution Preparation**: Identifies references to functions/variables defined in other object files or libraries.

# 4. Linker

- **Object File Reading** – Reads object files generated by the compiler/assembler.
- **Symbol Resolution** – Resolves references to functions and variables defined in other object files or libraries.
- **External Reference Resolution** – Connects calls/references to externally defined functions and variables.
- **Address Assignment** – Assigns final memory addresses to code and data sections.
- **Relocation** – Adjusts addresses in machine code according to the final memory layout.
- **Library Linking** – Searches and includes required code from static or shared libraries.
- **Dependency Resolution** – Ensures required modules and library components are included.
- **Duplicate Symbol Checking** – Detects conflicting or multiple definitions of the same symbol.
- **Unresolved Symbol Checking** – Reports symbols that are referenced but not defined.
- **Executable Generation** – Produces the final executable or shared library.
- **Entry Point Setup** – Identifies the starting address of the program for execution.

# 5. Loader

- Compilers, assemblers and linkers usually produce code whose memory references are made relative to an undetermined starting location that can be anywhere in memory.

  (relocatable machine code)

- The loader then puts together all of the executable object files into memory for execution.

- A loader calculates appropriate absolute addresses for these memory locations and amends the code to use these addresses.

# 6. Execute

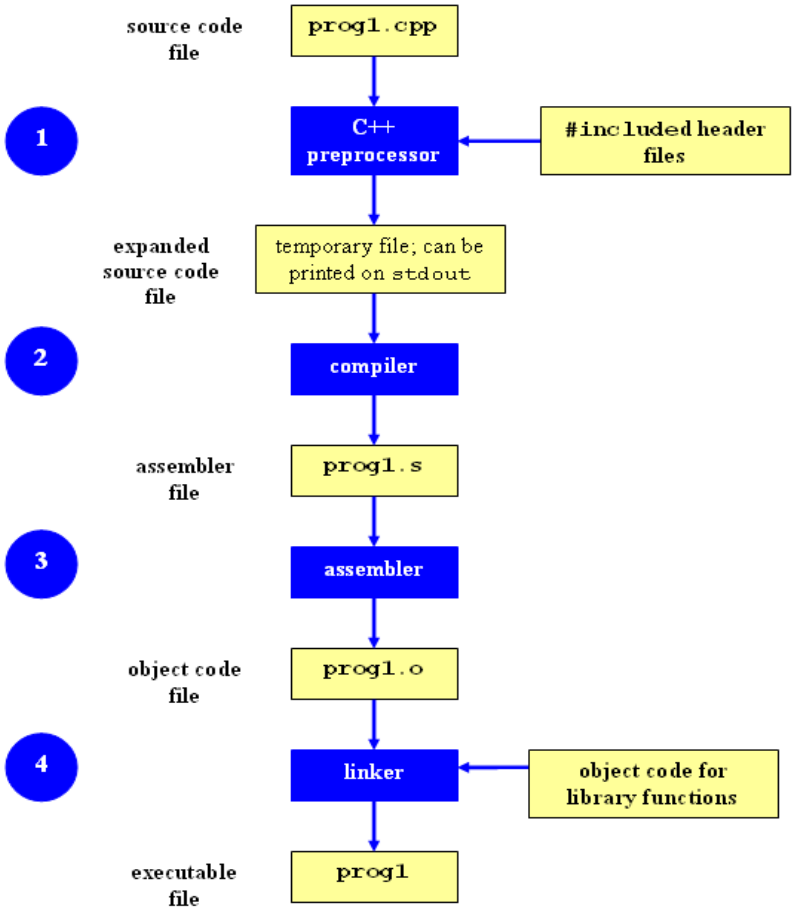
CPU executes the program one instruction at time.

# A Bit more about execution

- To stop the process after the pre-processor step, you can use the -E option:

  g++ -E prog.cpp –o prog.i

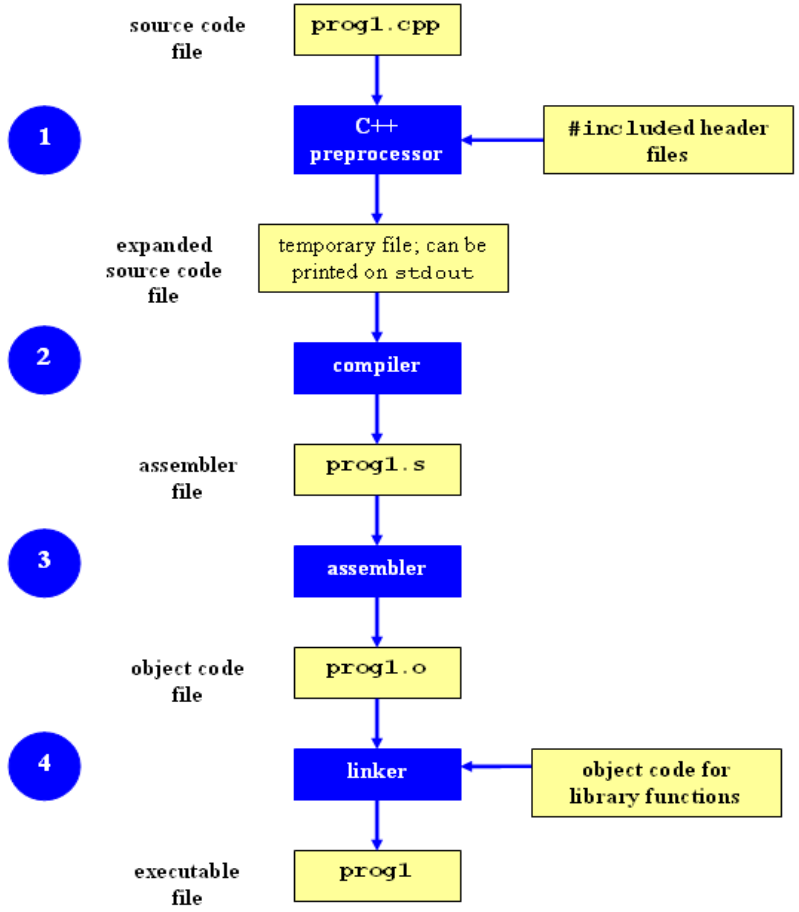
The expanded source code file will be printed on standard output (the screen by default).



2. To stop the process after the compile step, you can use the -S option:

g++ -S prog.cpp -o prog.s

By default, the assembler code for a source file named *filename.cpp* will be placed in a file named *filename.s*.



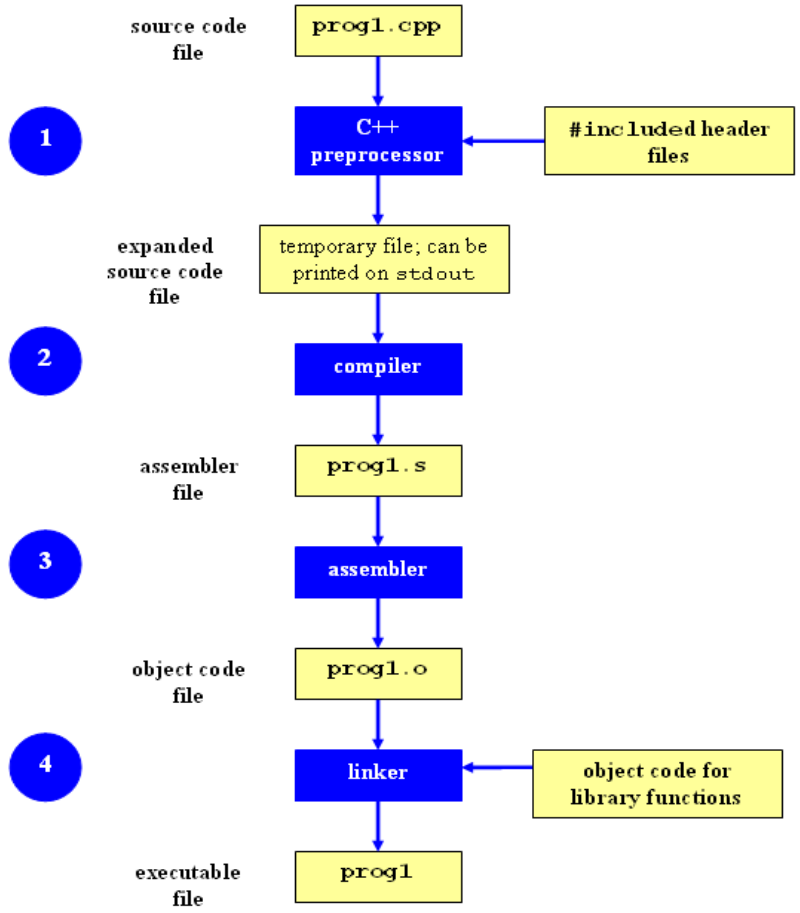
3. To stop the process after the assembly step, you can use -c option:

```
g++ -c prog.cpp -o prog.o
```

By default, the assembler code for a source file named *filename.cpp* will be placed in a file named *filename.o*.

```
g++ prog.o -o prog.exe

./prog.exe
```



## Command to view Object Code

objdump -d prog1.o

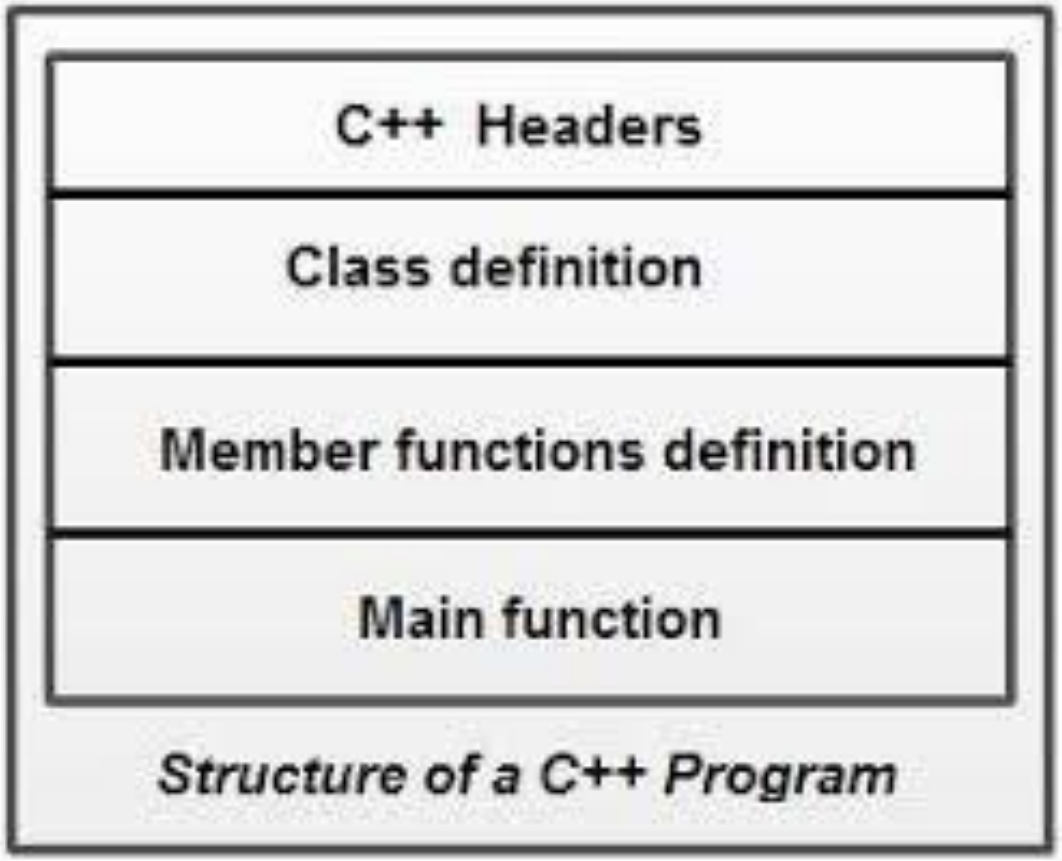
## Command to view executable file

objdump -d prog1.exe




# A simple C++ Program

# General Structure of C++ Program



# Input and Output in C++

User

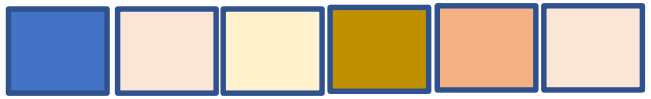
Console Output

Console Input

cout (ostream)

<< (insertion operator)

cin (istream)



>> (extraction operator)

Program

Programmer




# Basic program

```
#include<iostream>
using namespace std;
int main(){
cout<<"Hello world"<<endl;
}
```

# cin & cout

- cout works similar to printf
- cin works similar to scanf

**Example**

```
cout<<“Hello and welcome”;
cin>>variable;   //variable could be int, float, char array (without space)
```

# cout basics

- ‘\n’ is for new line, or you can use *endl*

```
cout << endl << “message”;
```

- ‘\t’ is for tab
- ‘\a’ is an alarm sound
- ‘\r’ is carriage return to go to the beginning of the current line

# Input char array with blanks

```
#include<iostream>
using namespace std;
int main(){
char str[20];
cin. getline(str,sizeof(str));
cout<<str<<endl;
}
```

# String basics - 1

```
int main(){
string name;
cout<<"Enter name: ";
getline(cin,name);  // cin>>name; will only take first word
if(name.compare("rocky sharma")==0)
 cout<<"Same"<<endl;
 else cout<<"Different"<<endl;
}
```

# String concat (connect)

```
int main(){
string name1,name2;
cout<<"string 1: "; cin>>name1;
cout<<"string 2: "; cin>>name2;
cout<<"String concat = "<<name1+", "+ name2;
}
// + means connect strings together
```




# Header files and namespace

# Header file and more

- #include <iostream>  // input-output stream for cin/cout
- using namespace std;

- Namespaces allow us to  differentiate same named  entities in various libraries. It is just a region of the code or library and not a function.
- ***std*** stands for standard I/O on the console screen.

# Need of namespace

```
#include <iostream>
int main() {
    int value;
    value = 0;
    double value; // Error here due to reuse of value
    value = 0.0;
}
```

# Example – namespace defines the ***scope***

```
#include <iostream>
using namespace std;
namespace ns1 { int value() {return 5;}}
namespace ns2 { int value() {return -5;}}

int main() {
cout << ns1::value() << '\n';  //5 will be displayed
cout << ns2::value() << '\n';     // -5 will be displayed
}
```

# What will be the output?

```
#include <iostream>
using namespace std;
namespace ns1 { int value() {return 5;}}
namespace ns2 { int x=10; int value() {return 4;}}
 int main() {
cout << ns1::value() << endl;
cout << ns2::value() << endl;
cout<< ns2::x<<endl;
}
```

# Without namespace could cause ::

```
#include<iostream>


int main(){
std::cout << “Hello there” << std::endl;
return 0;
}
```

# using namespace std;

- Thus using namespace std;  means cin/cout will be performed through standard console screen.

# Header file and namespace

- #include<iostream> includes all necessary files required of CIN and COUT opertions.
- using namespace std; allows us to reduce :: pollution for simpler programs

# Conclusion of the Namespace

**Namespace**: A namespace is a declarative region used to organize and group related identifiers such **as classes, functions, variables, and objects**.

**Purpose**: Prevents **name conflicts** when the same identifier is used in different parts of a program.

**Scope**: Names (identifiers) declared inside a namespace belong to that namespace and can be accessed using the namespace name.

**Syntax**: namespace NamespaceName { /* declarations */ }

**Accessing Members**: Use the **scope resolution operator ::**, e.g., NamespaceName::function().

**Using Directive**: using namespace NamespaceName; allows members to be used without repeatedly specifying the namespace name.

**Using Declaration**: using NamespaceName::member; brings only a specific member into the current scope.

# Conclusion of the Namespace *contd…*

**Nested Namespace**: A namespace can contain another namespace.

**Unnamed Namespace**: An unnamed namespace gives its members **internal linkage**, making them accessible only within the current translation unit.

**Standard Namespace:**C++ library components are generally defined in the **std namespace, e.g., std::cout, std::vector**.

**Namespace Alias**: A shorter alias can be created for a long namespace name, e.g., namespace ns = VeryLongNamespaceName;

**Benefit**: Improves **code organization, modularity, readability, and maintainability** while reducing naming collisions.

## Namespace with class

```
namespace College {
    class Student {
    public:
        void display() {
            cout << "Student";
        }
    };
}

College::Student s;
```

## Namespace with Function

```
namespace Math {
    int add(int a, int b) {
        return a + b;
    }
}

int x = Math::add(5, 3);
```

## Namespace with Object

```
namespace College {
    class Student {
    public:
        void display() {
            cout << "Student";
        }
    };

    Student s1;    // Object
}

College::s1.display();
```

**May I ask you to explore?**

- Namespace with Function Overloading
- Namespace with Inheritance
- Namespace with Static Data Member

# NAMESPACE CHALLENGE

Dear Curious Coder,

Think! Explore! Code! Repeat!

Namespaces are like rooms in a big building... same names can live peacefully in different rooms!

I know who you mean, just use me right!

**Same Name... Different Identity!**

## CHALLENGE:

Two show() functions, one name, zero confusion—how does C++ know whom to call? Investigate!

### SNEAK PEEK (What's Possible? 😎)

```
namespace A {  void show() { /* ... */ } }
namespace B {  void show() { /* ... */ } }
A::show();   // Calls A's show
B::show();   // Calls B's show
// No fight! No confusion!
```

### SAME NAME, DIFFERENT ROOM!

```
namespace A { int value = 10; }
namespace B { int value = 20; }
cout << A::value;  // 10
cout << B::value;  // 20
```


Same name, different identity!

### NESTED ROOMS!

```
namespace College {
  namespace CSE {
    class Student { ... };
  }
}
College::CSE::Student s;
```


Rooms inside rooms!

### USING – SHORTCUT PASS!

```
namespace Math { int add(int a,
int b) { return a+b; } }
using namespace Math;
cout << add(5, 3); // No Math::
```


Use less, code more!

### HIDE & SEEK!

```
namespace { int secret = 42; }
int main() {
  cout << secret; // Works here
}
// Secret can't be seen outside
// this file!
```


Unnamed namespace = Private!

### CONFLICT ALERT!

```
using namespace A;
using namespace B;
// Now 'show' alone is confused!
// Who to call? 😕
// Solve ambiguity like a pro!
```


Ambiguity is the real boss!

I make identical names behave differently without changing their names— that's my MAGIC! ✨

**YOUR MISSION** (Should you accept it 😉):
Explore. Experiment. Enjoy.
That's the C++ Way!

Understand it today...
Ace it in the exam!
Namespaces are not just names... they are SUPERPOWERS!

NO CONFUSION. JUST CLARITY. ONE NAME. MANY MEANINGS. HAPPY CODING!

Same struct… But Is It Really the Same?

# SAME NAME... DIFFERENT SUPERPOWERS!

## STRUCTURE IN C VS STRUCTURE IN C++

I'm classic C. Simple and straight!

I'm C++. Smarter & powerful!

### C PROGRAM

```c
#include <stdio.h>
#include <string.h>

struct Student {
   int rollNo;
   char name[30];
};

void display(struct Student s) {
   printf("Roll No: %d\n", s.rollNo);
   printf("Name: %s\n", s.name);
}

int main() {
   struct Student s1;

   s1.rollNo = 101;
   strcpy(s1.name, "Rahul");
   display(s1);
   return 0;
}
```

I FOLOW procedures! Functions are outside.

OUTPUT
Roll No: 101
Name: Rahul

### KEY DIFFERENCES

| C | | C++ |
|---|---|---|
| Primarily groups data | VS | Can group data + functions |
| Function defined outside the structure | VS | Member functions can be defined inside the structure |
| Declaration: struct Student s1; | VS | Declaration: Student s1; |
| Function call: display(s1); | VS | Function call: s1.display(); |
| C is procedure-oriented | VS | C++ is object-oriented |

### C++ PROGRAM

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

struct Student {
   int rollNo;
   string name;

   void display() {
      cout << "Roll No: " << rollNo << endl;
      cout << "Name: " << name << endl;
   }
};

int main() {
   Student s1;

   s1.rollNo = 101;
   s1.name = "Rahul";
   s1.display();
   return 0;
}
```

I bring data and behavior together!

OUTPUT
Roll No: 101
Name: Rahul

Language may be different, but the idea is to organize information smartly!

Same STRUCT word... Different world!
Think, Compare, Code like a PRO!

Learn the differences, Master the concepts, Build the future!

# Object Oriented Paradigm?

## 1. EVERY ENTITY IN THE WORLD

Everything we observe can be modeled as an **Entity**

**ENTITY**
(Living or Non-Living)

**ATTRIBUTES**
(State / Data)

They carry values to represent the existence of the entity uniquely in the world.

Example:
age, height, weight, color, speed, name, etc.

Behavior operates with attributes to represent the state of the entity over time in the world.

**BEHAVIORS**
(Actions / Methods)

The task for which they have been made or the process which can be applied on them.

Example:
aging(), run(), start(), calculate(), move(), etc.

**KEY IDEA**

Attributes describe WHAT an entity is.
Behaviors define WHAT an entity does and HOW it changes over time.

## 2. ATTRIBUTES & BEHAVIORS

General View of an Entity

**ATTRIBUTES**
(State / Data)

- Store values
- Describe the current state
- May change over time

Input / Stimulus
(From Environment)

Behaviors read or modify attributes and thus change the state of the entity.

Output / Result
(Change in State)

**BEHAVIORS**
(Actions / Methods)

- Perform actions
- Operate on attributes
- Cause state transitions

STATE OF THE ENTITY = Values of its Attributes at a particular time

## 3. EXAMPLE 1: LIVING ENTITY – CHILD

**CLASS : CHILD**

**ATTRIBUTES (State)**

- name = "Aarav"
- age = 0 months
- height = 50 cm
- weight = 3.2 kg
- bloodGroup = "O+"
- eyeColor = Black

**BEHAVIORS (Actions)**

- aging()
- heightGain()
- weightGain()
- eat()
- sleep()
- play()

**STATE TRANSITION OVER TIME**

**Birth** (age = 0 months)

age = 0 months
height = 50 cm
weight = 3.2 kg

aging()
heightGain()
weightGain()

**After 12 Months**

age = 12 months
height = 75 cm
weight = 9 kg

Behaviors operate on attributes and change the state of the child over time.

## 4. EXAMPLE 2: NON-LIVING ENTITY – CAR

**CLASS : CAR**

**ATTRIBUTES (State)**

- brand = Toyota
- color = Red
- speed = 0 km/h
- fuelLevel = 20 L
- gear = N (Neutral)
- isEngineOn = False

**BEHAVIORS (Actions)**

- start()
- accelerate()
- brake()
- refuel()
- stop()

**STATE TRANSITION OVER TIME**

Initial State

speed = 0 km/h
fuelLevel = 20 L
gear = N
isEngineOn = False

start()
accelerate()
accelerate()
...

After Some Time

speed = 40 km/h
fuelLevel = 19.8 L
gear = D
isEngineOn = True

Behaviors operate on attributes and change the state of the car over time.

## 5. RELATING TO OOP

| REAL WORLD | OOP CONCEPT |
|---|---|
| Entity | Object |
| Attributes | Data Members (Variables) |
| Behaviors | Member Functions (Methods) |
| State | Current Values of Attributes |
| Blueprint / Template | Class |

A Class is a blueprint. Objects are created from a class. Each object has its own attributes (state) and behaviors.

## 6. GOD ANALOGY & OBJECT LIFE CYCLE

A beautiful analogy to understand Object Life Cycle

**GENERATOR**
(Constructor)

Brings an object into existence and initializes its attributes.

**OPERATOR**
(Behavior / Methods)

The object performs actions (methods) and changes its state over time.

**DESTRUCTOR**
(Destructor)

Invoked when the object's lifetime ends and resources are released.

**OBJECT LIFE CYCLE**

Constructor()
(Object Creation)

↓

Object in Use
(Methods Execute)
(State Changes Over Time)

↓

Destructor()
(Object Deletion)

↓

Resources Released

**Note:** In C++, destructor releases resources such as dynamically allocated memory, file handles, network connections, etc. The memory occupied by the object itself is reclaimed as per how and where it was created (automatic or dynamic).

# C++ STRUCT — ENCAPSULATION & DATA HIDING

**BIND DATA, HIDE IT SMARTLY. ACCESS IT SAFELY.**

## EXAMPLE PROGRAM

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

struct Student {
 private:
    int marks;       // Data is hidden

 public:
    void setMarks(int m) {
        if (m >= 0 && m <= 100)
            marks = m;
    }

    void display() {
        cout << "Marks: " << marks << endl;
    }
};

int main() {
    Student s1;        // Object
    s1.setMarks(85);   // Allowed
    s1.display();

    // s1.marks = 95;  // ERROR: private data
    return 0;
}
```

**Try this!**
Can we access marks directly from main()?
?


## KEY POINTS FOR STUDENTS

**1. ENCAPSULATION** — Data and the functions that work on that data are wrapped together inside one unit.

**2. DATA HIDING** — Data members declared as **private** cannot be accessed directly from outside the structure.

**3. CONTROLLED ACCESS** — Public member functions act as **gatekeepers** to read or modify the private data safely.

**4. OBJECT** — s1 is an **object** of the struct Student. We interact with data through its member functions.

**5. IMPORTANT** — Even in C++, **struct** supports both **encapsulation** and **data hiding**.

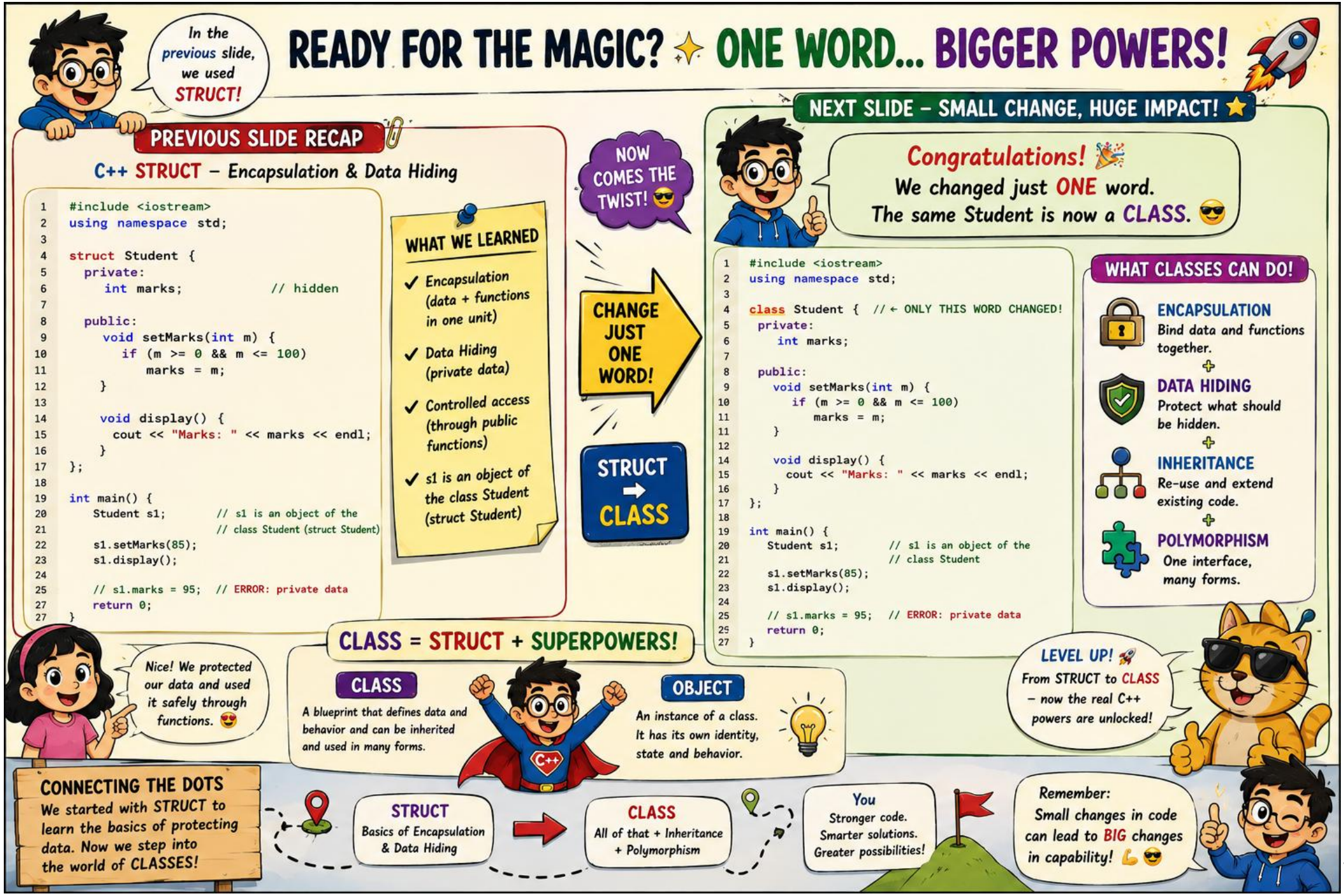
## WHY IS THIS USEFUL?

- Prevents accidental changes to data.
- Improves security and reliability.
- Makes code modular and maintainable.

## TAKEAWAY

Encapsulate what varies.
Hide what shouldn't be touched.
Expose what is necessary.
That's **smart programming!**

# Week-2
# Classes and Objects


In the previous slide, we used STRUCT!
READY FOR THE MAGIC? ONE WORD... BIGGER POWERS!
PREVIOUS SLIDE RECAP
C++ STRUCT – Encapsulation & Data Hiding
#include <iostream>
using namespace std;
struct Student {
private:
int marks; // hidden
public:
void setMarks(int m) {
if (m >= 0 && m <= 100)
marks = m;
}
void display() {
cout << "Marks: " << marks << endl;
}
};
int main() {
Student s1; // s1 is an object of the
// class Student (struct Student)
s1.setMarks(85);
s1.display();
// s1.marks = 95; // ERROR: private data
return 0;
}
WHAT WE LEARNED
Encapsulation (data + functions in one unit)
Data Hiding (private data)
Controlled access (through public functions)
s1 is an object of the class Student (struct Student)
NOW COMES THE TWIST!
CHANGE JUST ONE WORD!
STRUCT → CLASS
NEXT SLIDE – SMALL CHANGE, HUGE IMPACT!
Congratulations!
We changed just ONE word.
The same Student is now a CLASS.
#include <iostream>
using namespace std;
class Student { // ← ONLY THIS WORD CHANGED!
private:
int marks;
public:
void setMarks(int m) {
if (m >= 0 && m <= 100)
marks = m;
}
void display() {
cout << "Marks: " << marks << endl;
}
};
int main() {
Student s1; // s1 is an object of the
// class Student
s1.setMarks(85);
s1.display();
// s1.marks = 95; // ERROR: private data
return 0;
}
WHAT CLASSES CAN DO!
ENCAPSULATION
Bind data and functions together.
DATA HIDING
Protect what should be hidden.
INHERITANCE
Re-use and extend existing code.
POLYMORPHISM
One interface, many forms.
Nice! We protected our data and used it safely through functions.
CLASS = STRUCT + SUPERPOWERS!
CLASS
A blueprint that defines data and behavior and can be inherited and used in many forms.
OBJECT
An instance of a class. It has its own identity, state and behavior.
LEVEL UP!
From STRUCT to CLASS – now the real C++ powers are unlocked!
CONNECTING THE DOTS
We started with STRUCT to learn the basics of protecting data. Now we step into the world of CLASSES!
STRUCT
Basics of Encapsulation & Data Hiding
CLASS
All of that + Inheritance + Polymorphism
You
Stronger code.
Smarter solutions.
Greater possibilities!
Remember:
Small changes in code can lead to BIG changes in capability!

# FUNCTIONS DEFINED OUTSIDE A C++ CLASS

**"Declare inside… Define outside!"**

## CLASS + OUTSIDE DEFINITIONS

```
class Student {
private:
    int marks;

public:
    void setMarks(int m);    // Declaration
    void display();          // Declaration
};

// Definitions outside the class

void Student::setMarks(int m) {
    marks = m;
}

void Student::display() {
    cout << "Marks: " << marks << endl;
}
```

## WHY `::` ?

```
Student::display()
          ↑
Scope Resolution Operator Tells C++
that display() belongs to Student.
```

## WHY DEFINE OUTSIDE?

- Cleaner class definition
- Better readability
- Easier maintenance
- Useful for large projects
- Separates interface & implementation

**⋆ REMEMBER  Inside class → Declaration / Interface  |  Outside class → Definition / Implementation**

# ⚡ INLINE FUNCTION IN C++

"Small function? Skip the function-call overhead!"

## Example

```
inline int square(int n) {
    return n * n;
}

int main() {
    cout << square(5);
}
```

## WHAT DOES inline MEAN?

- Requests the compiler to expand the function at the call site
- Can reduce function-call overhead
- Best suited for small, frequently called functions
- Request, NOT a guarantee; the compiler may ignore it

## 🔍 WITHOUT inline

**Call → Function → Return**

↕

**Call overhead**

## ⚡ WITH inline

**Call → Function code inserted here**

↓

**Less overhead**

**⋆ REMEMBER: inline ≠ guaranteed expansion | Small + frequently called → good candidate**

# Example of Inline

```
//Code
#include <iostream>
using namespace std;
inline int max(int a, int b){
    return a>b ? a : b;
}
int main(){
    cout << max(10, 20);
    cout << " " << max(99, 88);
    return 0;
}
```

```
//Compiler Interpretation
//Compiler Interpretation
#include <iostream>
using namespace std;
int main(){
    cout << (10>20 ? 10 : 20);
    cout << " " << (99>88 ? 99 :88);
    return 0;
}
```

# Characteristics of Inline Functions

- Inline functions allow you to create very efficient code.
- Like the register specifier, inline is actually just a request, not a command, to the compiler.
- If the function definition is too long or complex, the compiler can choose to ignore it. For example, inline functions may not work for functions having static variables or recursive functions.

# Inline member functions

```
#include <iostream>
using namespace std;
class myclass {
    int a, b;
public:
    void init(int i, int j);
    void show();
};
// Create an inline function.
inline void myclass::init(int i, int j){
    a = i;
    b = j;
}
// Create another inline function.
inline void myclass::show(){
    cout << a << " " << b << "\n";
}
int main(){
    myclass x;
    x.init(10, 20);
    x.show();
    return 0;
}
```

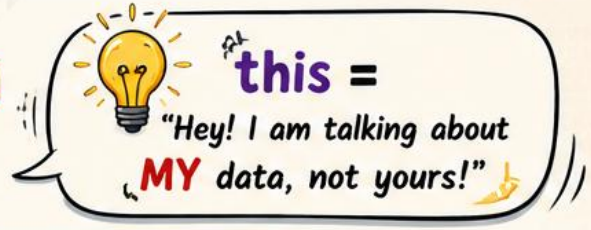
# Defining Inline Functions within a class

- When a function is defined inside a class declaration, it is automatically made into an inline function (if possible).
- It is not necessary (but not an error) to precede its declaration with the inline keyword.

```
#include <iostream>
using namespace std;
class myclass {
    int a, b;
public:
    // automatic inline
    void init(int i, int j) { a=i; b=j; }
    void show() { cout << a << " " << b << "\n"; }
};
int main(){
    myclass x;
    x.init(10, 20);
    x.show();
    return 0;
```

# this POINTER IN C++

## Who Owns This Data?




### EXAMPLE

```
1  #include <iostream>
2  using namespace std;
3
4  class Student {
5   private:
6      int marks;
7
8   public:
9      void setMarks(int marks) {
10         this->marks  =  marks;    // object's data = parameter
11     }
12
13     void display() {
14         cout << "Marks:  " << this->marks << endl;
15     }
16 };
17
18 int main() {
19     Student s1;        // s1 is an object of class Student
20     s1.setMarks(85);   // this points to s1
       s1.display();
21     return 0;
22 }
```

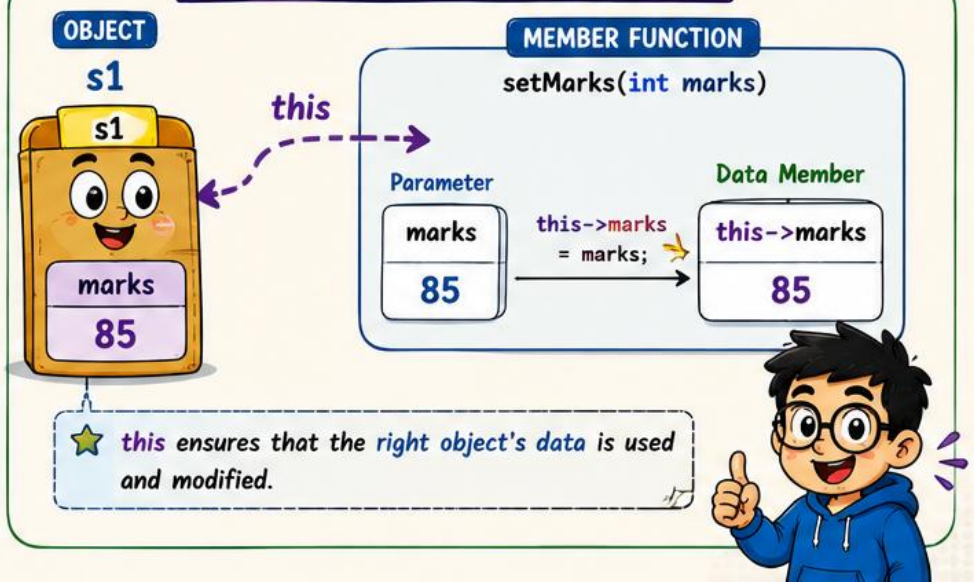
this is an implicit pointer available inside non-static member functions.

When these functions are called using s1, this points to s1.

### WHAT IS 'this'?

- this is a pointer to the current object.
- It helps differentiate between data members and parameters having the same name.
- this->marks refers to the object's data member.
- It makes the code clear, safe and unambiguous.

### VISUAL REPRESENTATION




### REMEMBER THIS!

- this points to the object that calls the function.
- this-> is used to access the object's members.
- Removes ambiguity when names are the same.
- Makes your code robust, readable and powerful!

# POINTER & REFERENCE → NOW MEET OBJECTS!

## 1. POINTER TO A VARIABLE

```
int x = 10;
int *p = &x;
```

p stores the address of x

**Pointer = ADDRESS**

## 2. REFERENCE TO A VARIABLE

```
int x = 10;
int &r = x;
```

r is another name for x

**Reference = ALIAS**

## ⚡ WHY USE THEM?

- Avoid unnecessary copying
- Modify original data
- Pass data efficiently
- Useful in dynamic data structures

**OOP CONNECTION: IF A VARIABLE CAN HAVE A POINTER / REFERENCE… CAN AN OBJECT HAVE ONE TOO?**

## OBJECT + POINTER + REFERENCE

```
Student s1;

Student *p = &s1;   // Pointer to Object
Student &r = s1;    // Reference to Object
```

## THREE WAYS - SAME OBJECT

```
s1.display();       // Original object
p->display();      // Through pointer
r.display();        // Through reference
```

**REMEMBER** Variable → Pointer / Reference | Object → Pointer / Reference to Object

# More details on Class and Objects

# Specifying a Class

- A class is a way to bind the data and its associated functions together.
- A class specification has 2 parts:
  - Class declaration
  - Class function definitions

- **Class declaration**: describes type and scope of its members.
- **Class function definitions**: describes how the class functions are implemented.

# General form of Class declaration

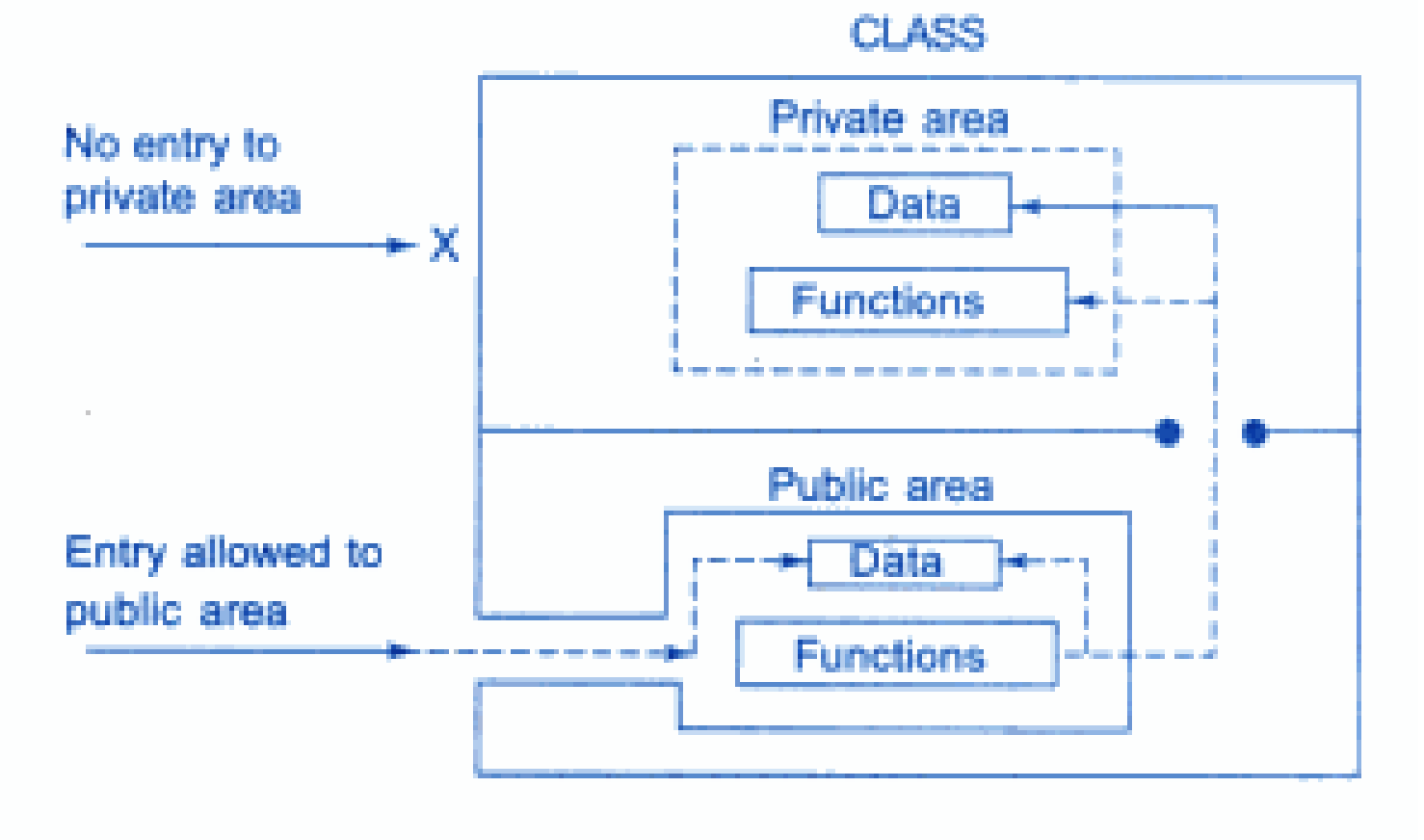
```
class  class_name
{
    private:
        variable declarations;
        function declarations;
    public:
        variable declarations;
        function declarations;
} ;
```

data members

member functions

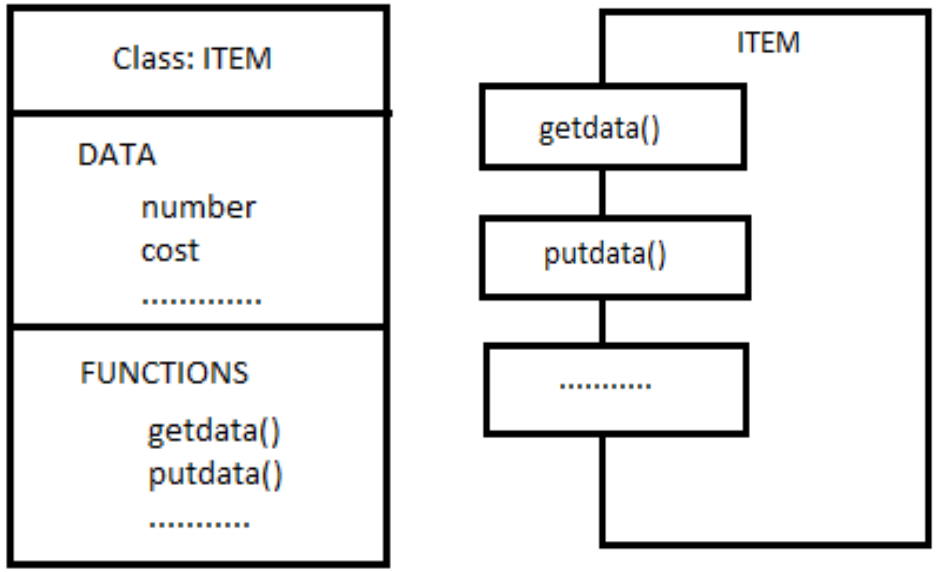
class members

# Data Hiding


CLASS
Private area
Data
Functions
No entry to
private area
X
Public area
Data
Functions
Entry allowed to
public area

# A Simple Class Example

```
class item
{
        int number;
        float cost;
    public :
        void getdata(int a, float b);
        void putdata (void);
};
```




**Class Representation**

**Creating Objects:**

item x;

**Accessing Class Members:**

Object_name.function-name (actual-arguments);

x.getdata(100, 75.5);  x.putdata( );

# Creating Objects

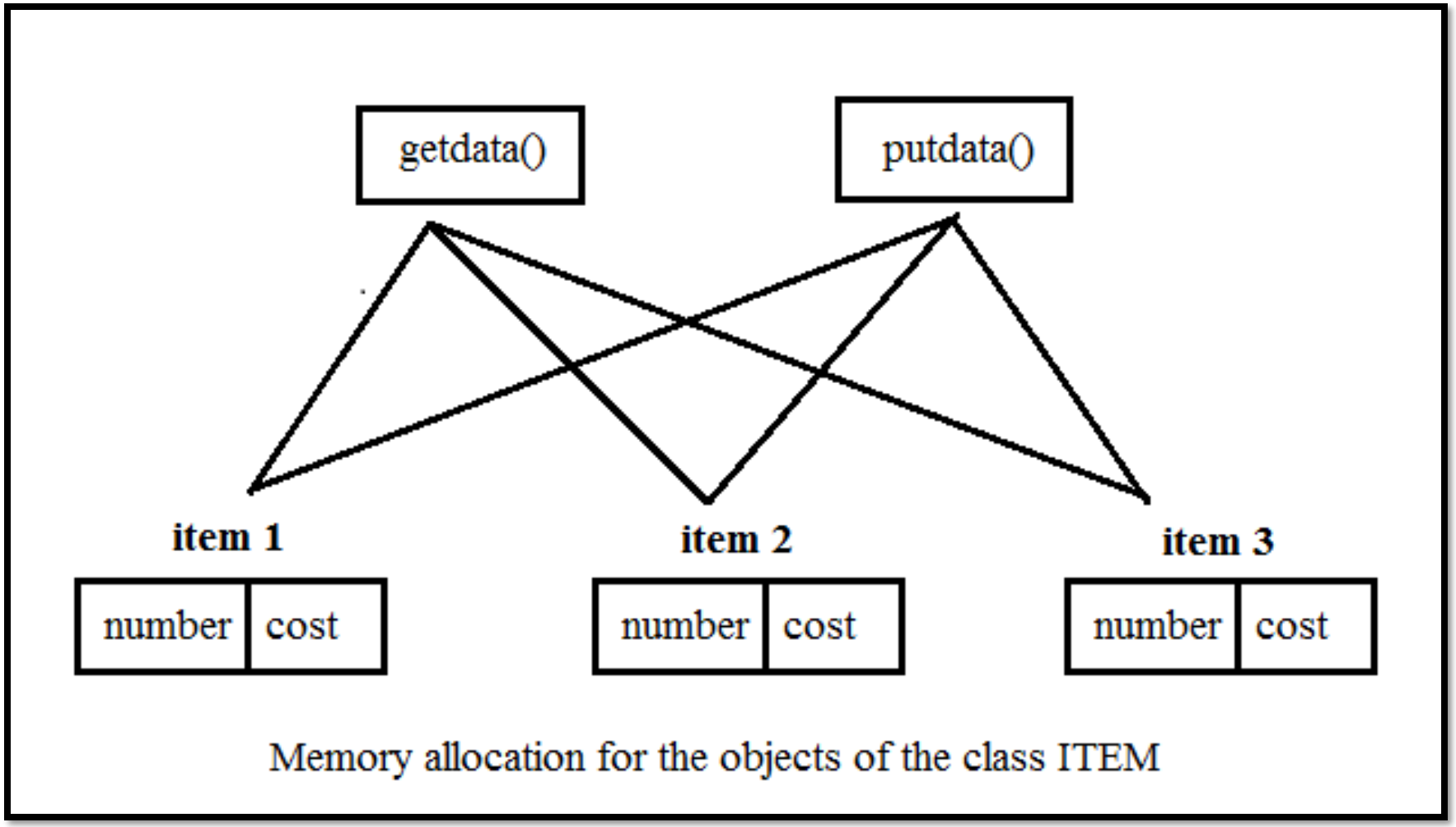
- Objects can also be created as follows:

```
class item
{
        .....
        .....
        .....
} x, y, z;
```

# Memory Allocation for Objects

- Memory space for objects is allocated when they are declared and not when the class is specified : **partially true.**
- Since all the objects belonging to that class use the same member functions, no separate space is allocated for member functions when the objects are created.
- Only space for member variables is allocated separately for each object. Because, member variables will hold different data values for different objects.

# Example




Memory allocation for the objects of the class ITEM

# Defining Member Functions

- **Outside the class definition**

  When function is defined outside class, it requires a prototype declaration in the class.

```
return-type class-name :: function-name (argument declaration)
{
    Function body
}
```

```
void item :: getdata (int a, int b)
{
        number = a;
        cost = b;
}
```

- **Inside the class definition**

  When function is defined inside class, it does not require a prototype declaration in the class.

```
class item
{
        int number;
        float cost;
    public:
        void getdata(int a, float b)
        {
                number=a;
                cost=b;
        }
};
```

# Nesting of Member Functions

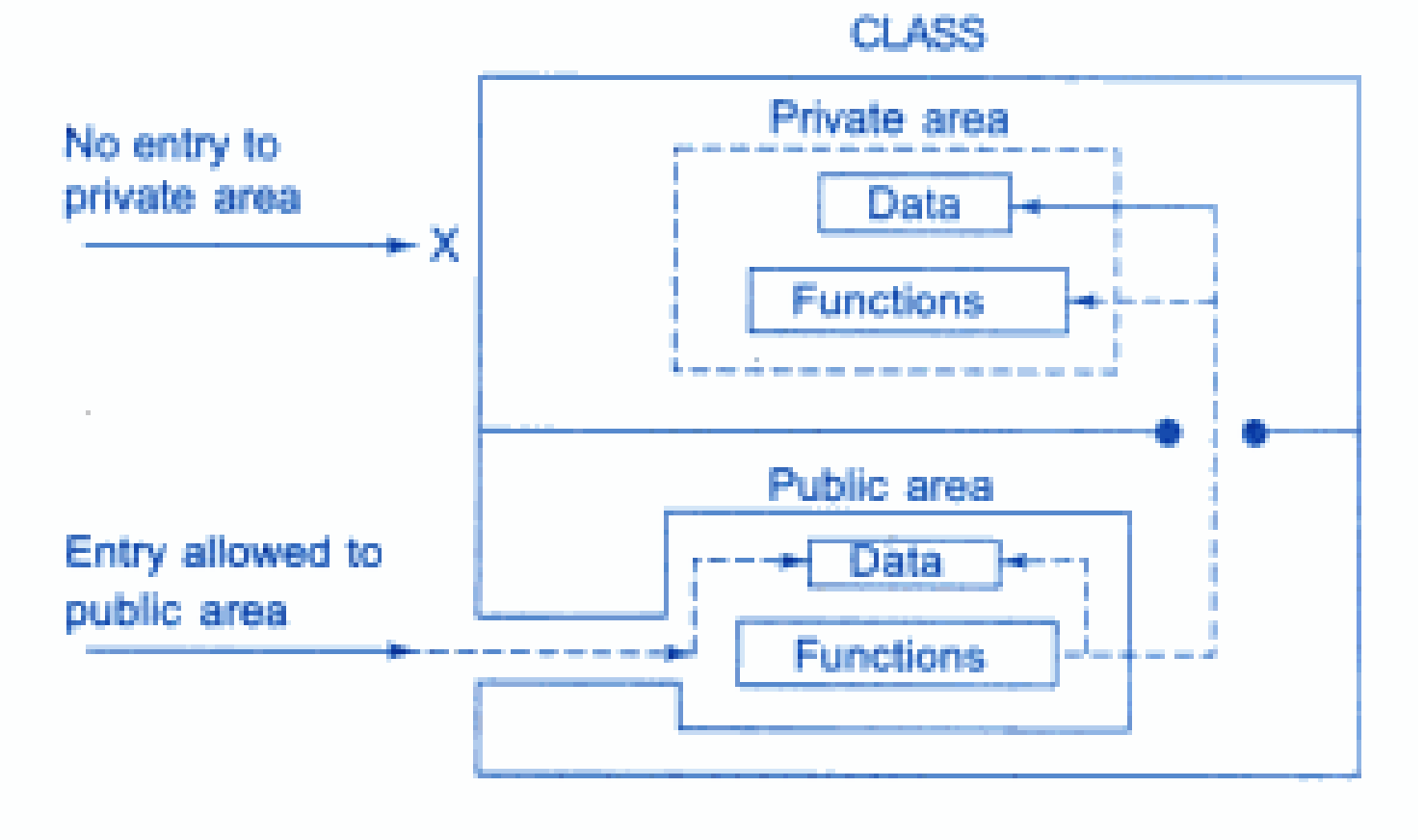
- An object of a class using dot operator generally calls a member function of the class.
- However, a member function may also be called by using its name inside another member function of the same class.
- This is known as “Nesting of Member Functions”

# Nesting of Member Functions

```
class emp
{
        float basic;
public:
        void display()
                { cout<<basic; }
        void takedata()
          {    cin>>basic;
               display();
          }
};
 void main()
{
          emp e1;
          e1.takedata();
}
```

A member function of a class can be called by another member function Of the same class. takedata() is calling display()

# Data Hiding Revisited



# Access Specifiers in C++

3 types of access modifiers available in C++:

- **Public**
- **Private (by default)**
- **Protected**

# Access Specifier: Public

- Public members are accessible outside the class.
- Public members are accessible to both member Functions and non-member Functions.
- Objects can access the members directly using dot operator. E.g.
  - object name.public data member
  - object name.public member function

# Access Specifier: Public

```
class emp
{   int empno;
    char empname[20];
    float basic;
public:
  char address[20];
        void readdata()
        {       cin>>empno;
                gets(empname);
                gets(address);
                cin>>basic;
        }
        void displaydata()
        {       cout<<empno;
                cout<<empname;
                cout<<basic<<address;
        }
};// end of class
```

emp e1;
Outside the class we can access

e1.address //OK
Object name.public data member

e1.readdata() //OK
Object name.public member function

e1.displaydata()//OK
Object name.public member function

# Access Specifier: Private

- Private Members can only be accessed by the member functions of that class.
- They cannot be accessed by any non member functions (data is hidden from outside world).
- Object of a class cannot access private members using dot operators.
- All data at the beginning of a class is by default private, until any specifier is mentioned.

# Access Specifier: Private

```
class emp
{   int empno;
    char empname[20];
    float basic;
public:
        void readdata()
        {       cin>>empno;
                gets(empname);
                cin>>basic;
        }
        void displaydata()
        {       cout<<empno;
                cout<<empname;
                cout<<basic;
        }
};// end of class
```

```
void main()
{
   emp e1;
   //To input details
   cin>>e1.empno; //NOT OK

   e1.readdata(); //OK
   //To Display details
   e1.displaydata();// OK
}
```

# Private Member Functions

```
class sample
{
        int m;
        void read (void);
    public:
        void update (void);
        void write (void);
};

if s1 is an object of sample, then

        s1.read();              //won't work; objects cannot access
                                //private members
```

# Private Member Functions

**s1.read()**

is illegal. However, the function **read()** can be called by the function **update()** to update the value of **m.**

```
void sample : : update ( void )
{
        read();         //simple call; no object used
}
```

# Magic of Static Keyword

# 1. STATIC DATA MEMBER

**One copy • Shared by every object**

```
class Student {
public:
    int rollNo;
    static int count;

    Student() {
        count++;
    }
};

int Student::count = 0;
```

**Non-static:**
**s1 → rollNo**
**s2 → rollNo**
**s3 → rollNo**

**Static:**
**Student::count**


```
Student s1;
Student s2;
Student s3;

cout << Student::count;    // 3
```

**One shared copy**

**Use when data represents the CLASS as a whole, e.g., total objects created.**

**REMEMBER: normal data member → one per object | static data member → one per class**

# 2. STATIC MEMBER FUNCTION

**Belongs to the class • No object required**

```
class Student {
    static int count;

public:
    static void showCount() {
        cout << count;
    }
};

Student::showCount();
```

## KEY RULES

✓ Called using class name

✓ Can directly access static members

✗ Cannot directly access non-static members

✓ No `this` pointer

**Why? → Operation belongs to the CLASS, not one particular object.**

**Student::showCount(); → "Ask the CLASS, not an individual object."**

# 3. STATIC OBJECT

**Created once • Retains state • Lives until program ends**

```
void test() {
    static Student s1;

    // use s1
}



test();  → create s1

test();  → same s1

test();  → same s1
```

## WHAT MAKES IT SPECIAL?

- Constructed only once
- State persists between calls
- Not destroyed when the function ends
- Destroyed when the program terminates

**Useful when a local object must remember its previous state.**

**STATIC LOCAL OBJECT ≠ STATIC DATA MEMBER | One object persists vs. one shared class member**

# COMBINED PROGRAM

## Static data member + static member function + static object

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

class Student {
    int rollNo;
    static int count;

public:
    Student(int r) : rollNo(r) { count++; }

    static void showCount() {
        cout << "Total: " << count << endl;
    }

    void show() {
        cout << "Roll: " << rollNo << endl;
    }
};
int Student::count = 0;
void demo() {
    static Student s4(104);   // static object
    s4.show();
}

int main() {
    Student s1(101), s2(102);
    Student::showCount();
    demo();
    demo();
    Student::showCount();
}
```

### TRACE THE `static`

① count ONE shared class variable

② showCount() Called through Student::

③ s4 ONE static object
Same object on both demo() calls

④ Final count 3 objects total



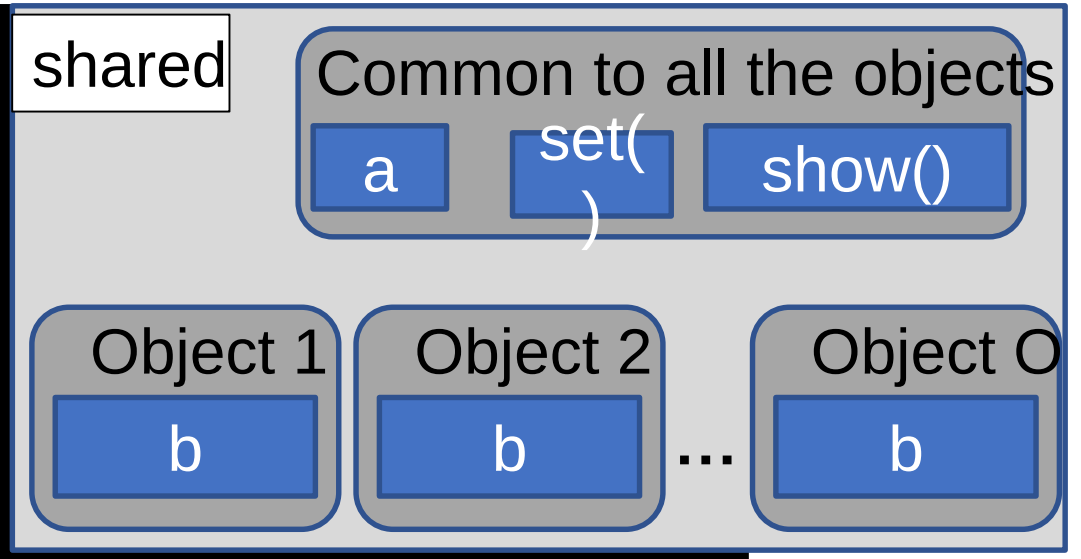

# More on Static Keyword

# Static Data Member

- Single copy exists; shared by all the objects.
- Also known as class variables.
- Exists and initialized to zero before any object is created.
- Within a class, they are declared not defined.
- Public static data member can be accessed directly by using scope resolution operator.
- Requires a global definition outside the class.
  - Re-declaration using scope resolution operator.

# Example 1

```
#include <iostream>
using namespace std;
class shared {
    static int a;
    int b;
public:
    void set(int i, int j) {a=i; b=j;}
    void show();
} ;
int shared::a; // define a
void shared::show(){
    cout << "This is static a: " << a;
    cout << "\nThis is non-static b: " << b;
    cout << "\n";
}
```



# Example 1

```
int main(){
    shared x, y;
    x.set(1, 1); // set a to 1
    x.show();
    y.set(2, 2); // change a to 2
    y.show();
    x.show();
    return 0;
}
```

**Output**

This is static a: 1

This is non-static b: 1

This is static a: 2

This is non-static b: 2

This is static a: 2

This is non-static b: 1

# Example 2

```
#include <iostream>
using namespace std;
class shared {
public:
    static int a;
} ;
```

```
int main(){
    // initialize a before
creating any objects
    shared::a = 99;//public
    cout << "This is initial
value of a: " << shared::a;
    cout << "\n";
    shared x;
    cout << "This is x.a: " <<
x.a;
    return 0;
}
```

# Uses

- Provide access control of shared resource used by all the objects of a class, e.g. writing a file.
- To keep track of the number of objects of a particular class type.
- Note:

Virtually eliminates any need for global variables.

# Static Member Functions

- Can access only other static members of the class.
- Do not have a **this** pointer.
- They cannot be declared as **const** or **volatile**.
- They may not be **virtual**.
- There cannot be **static** and **non-static** version of the same function.
- Can be called using
  - An object.
  - Class name and scope resolution operator.

# Example

- **Objective: Write a program to provide access control to some shared resource.**

Example:

Create several objects, each of which wants to print a file on a specific printer. Clearly, only one object can be allowed to print a file at a time. In this case, declare a **static** variable that indicates when the printer is in use and when it is free. Each object then interrogates this variable to get the printer.

# Contd…

```
#include<iostream>
using namespace std;
class shared{
    static int resource;
public:
    static int getResource(){
        if(resource)
            return 0;
        else{
            resource = 1;
            return 1;
        }
    }
    void freeResource(){
        resource = 0;
    }
};
```

```
int shared :: resource;
int main(){
   shared o1, o2;
   if(o1.getResource())
        cout << "\no1 has resource.";
   if(!shared :: getResource())
        cout << "\no2 access denied.";
   o1.freeResource();
   if(shared :: getResource())
        cout << "\no2 has resource.";
   return 0;
}
```

# Uses

- Very limited application.
- Pre-initialize **private static** data members of a class before any object is actually created.

# Example

```
#include <iostream>
using namespace std;
class static_type {
    static int i;
public:
    static void init(int x) {
        i = x;
    }
    void show() {
        cout << i;
    }
};
int static_type::i; // define i
int main(){
// init static data before object
creation
    static_type::init(100);
    static_type x;
    x.show(); // displays 100
    return 0;
}
```

# Dynamic memory allocation

# Dynamic Memory Allocation in C++

**Why do we need memory at run time?**

**Normally:**

```
int x = 10;
```

Memory is automatically managed.

**But what if the size is known only at run time?**

```
int n;
cin >> n;
```

Now we may need memory for n integers.

## 👉 Dynamic memory allocation

Allocate memory when the program is running, rather than fixing the required size beforehand.

**This becomes especially useful when the amount of data is unknown until execution.**

# new`- Create Memory at Run Time

**Think: new → “Give me memory from the heap.”**

```
int *p = new int;
```

**We can then store a value:**

```
*p = 25;

cout << *p;        // 25
```

**Key idea**

**new**

↓

**“Give me memory from the heap.”**

The pointer stores the address of the dynamically allocated memory.

# Dynamic Memory Initialization

**Allocate memory + initialize it at run time**

```
int *p = new int;

*p = 25;

delete p;
```

**Complete program:**

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

int main() {
    int *p = new int;
    *p = 25;
    cout << *p;
    delete p;
    return 0;
}
```

**But C++ also allows:**

```
int *p = new int(25);
```

new int(25)

- Allocate memory
- Initialize it with 25

⋆ **This is dynamic initialization.**

# Dynamic Memory Allocation for an Array

**What if we need MANY values?**

**Suppose:**

```
int n;
cin >> n;
```

**We don't know the number of elements before the program runs.**

**Instead of:**

```
int arr[100];
```

**`new[]` → Dynamic Array**

```
int *arr = new int[n];
```

The array size is decided at run time.

**Memory allocated with `new[]` must be released with `delete[]`.**

```
new int[n]   ⇔   delete[] arr
```

# Complete Dynamic Array Program

**Input → Allocate → Use → Release**

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

int main() {

    int n;
    cout << "Enter number of students: ";
    cin >> n;

    int *marks = new int[n];

    for (int i = 0; i < n; i++) {
        cin >> marks[i];
    }

    for (int i = 0; i < n; i++) {
        cout << marks[i] << " ";
    }

    delete[] marks;

    return 0;
}
```

## Execution Flow

① Read n

② new int[n]

③ Use marks[i]

④ delete[] marks

# Dynamic Initialization + Dynamic Array

## Single value

```
int *p = new int(25);
```

**Allocation + initialization**

new int(25)

new → memory
25 → initial value

## Array

```
int *arr = new int[5]{10, 20, 30, 40, 50};
```

**Here:**

new[] → Dynamic memory allocation
{...} → Initialization

**For run-time size:**

```
int *arr = new int[n];
```

# Now Connect It to OOP

**What if the data type is a class?**

```
class Student {
public:
    int rollNo;

    void display() {
        cout << rollNo;
    }
};
```

**Now create dynamically:**

```
Student *p = new Student;

p->rollNo = 101;
p->display();

delete p;
```

## Dynamic Object

```
Student *p = new Student;
```

We have dynamically created a Student OBJECT.

```
p->rollNo
p->display()
```

delete p; → release the object

# Complete Dynamic Object Program

A class object can also live in dynamically allocated memory

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

class Student {
public:
    int rollNo;

    void display() {
        cout << "Roll No: " << rollNo << endl;
    }
};

int main() {

    Student *p = new Student;

    p->rollNo = 101;
    p->display();

    delete p;

    return 0;
}
```

**What happens?**

new Student
↓
Create object
↓
p->rollNo
↓
p->display()
↓
delete p
↓
Release object

# Dynamic Array of Objects

**What if we need MANY Student objects?**

If the number of students is known only at run time:

```
int n;
cin >> n;
```

**Then:**

```
Student *students = new Student[n];
```

## Dynamic Array of Objects

```
for (int i = 0; i < n; i++) {
    cin >> students[i].rollNo;
}

for (int i = 0; i < n; i++) {
    students[i].display();
}

delete[] students;
```

**new[] ↔ delete[]**

# The Complete Learning Chain

**From a variable to a dynamically allocated array of objects**

```
VARIABLE
    ↓
 int x;

    ↓

POINTER
    ↓
int *p;

    ↓

DYNAMIC MEMORY
    ↓
int *p = new int;

    ↓

DYNAMIC INITIALIZATION
    ↓
int *p = new int(25);

    ↓

DYNAMIC ARRAY
    ↓
int *arr = new int[n];

    ↓

DYNAMIC OBJECT
    ↓
Student *p = new Student;

    ↓

DYNAMIC ARRAY OF OBJECTS
    ↓
Student *s = new Student[n];
```

🎯 **Each step extends the same idea: memory is obtained when the program needs it.**

# Golden Rules

**The `new` / `delete` pairs you must remember**

## SINGLE OBJECT / VALUE

```
new       → Get memory
delete    → Release one object
```

```
int *p = new int;
...
delete p;
```

## ARRAY

```
new[]     → Get memory for an
            array
delete[]  → Release an array
```

```
int *arr = new int[n];
...
delete[] arr;
```

**new -> Get memory | delete -> Release one object | new[] -> Get array memory | delete[] -> Release array**

# Final OOP Connection

**A class is a user-defined data type**

### Built-in type

```
int *p = new int;
```

↓

**Dynamic int**

### Class type

```
Student *p = new Student;
```

↓

**Dynamic Object**

### Array of objects

```
Student *s = new
Student[n];
```

↓

**Dynamic collection of objects**

**The same memory-management mechanism works with built-in types, single objects, and arrays of objects.**

**Dynamic memory is the bridge from basic pointers to dynamic OOP data structures.**

# Scope Resolution Operator

1. Member functions defined outside class

- Using Binary scope resolution operator (::)
- “Ties” member name to class name
- Uniquely identify functions of particular class
- Different classes can have member functions with same name
- Syntax for defining member functions

```
returnType className::MemberFunctionName( )
{
                    …
}
```

# Scope Resolution Operator contd…

## 2. **To access a global variable when there is a local variable with same name**

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

int x;  // Global x

int main() {
  int x = 10; // Local x
  cout << "Value of global x is " << ::x;
  cout << "\nValue of local x is " << x;
  return 0;
}
```

# Scope Resolution Operator contd…

## 3. **To access a class's static variables**

```
class Test {
   static int x;
public:
   static int y;
   void func(int x)      {  // We can access class's static variable even if there is a local variable
     cout << "Value of static x is " << Test::x;
     cout << "\nValue of local x is " << x;
                 }
};
int Test::x = 1;          // In C++, static members must be explicitly defined like this
int Test::y = 2;
int main() {
   Test obj;
   int x = 3 ;
   obj.func(x);
   cout << "\nTest::y = " << Test::y;
   return 0;
}
```

# Scope Resolution Operator contd…

4. **In case of multiple Inheritance**

If same variable name exists in two ancestor classes, we can use scope resolution operator to distinguish.

Detail would be discussed in inheritance……

# Scope Resolution Operator contd…

## 5. Including In-built libraries

```
// Create a GradeBook object and call its displayMessage function.
#include <iostream>
using std::cout;
using std::cin;
using std::endl;

#include <string> // program uses C++ standard string class
using std::string;
using std::getline;

// GradeBook class definition
class GradeBook
{
public:
   // function that displays a welcome message to the GradeBook user
   void displayMessage( string courseName )
   {
      cout << "Welcome to the grade book for\n" << courseName << "!"
         << endl;
   } // end function displayMessage
}; // end class GradeBook

// function main begins program execution
int main()
{
   string nameOfCourse; // string of characters to store the course name
   GradeBook myGradeBook; // create a GradeBook object named myGradeBook
```

# Scope Resolution Operator contd…

## 5. **Including In-built libraries.**

```
31    // prompt for and input course name
32    cout << "Please enter the course name:" << endl;
33    getline( cin, nameOfCourse ); // read a course name with blanks
34    cout << endl; // output a blank line
35
36    // call myGradeBook's displayMessage function
37    // and pass nameOfCourse as an argument
38    myGradeBook.displayMessage( nameOfCourse );
39    return 0; // indicate successful termination
40 } // end main
```

```
Please enter the course name:
CS101 Introduction to C++ Programming

Welcome to the grade book for
CS101 Introduction to C++ Programming!
```

# Nested Classes

- A class is declared within another class.
- The outer class is called enclosing class and inner class is called the nested class.
- A nested class is a member and as such has the same access rights as any other member.
- The members of an enclosing class have no special access to members of a nested class; the usual access rules shall be obeyed.

# Nested Classes Example

```
#include<iostream>
using namespace std;
class Enclosing                          /* start of Enclosing class declaration */
{
      int x;
      class Nested
      {                                  /* start of Nested class declaration */
            int y;
             void NestedFun(Enclosing e)
             {
                  cout<<e.x;           // works fine: nested class can access
                                       // private members of Enclosing class
             }
   };                                    // declaration Nested class ends here
};                                       // declaration Enclosing class ends here
int main(){}
```

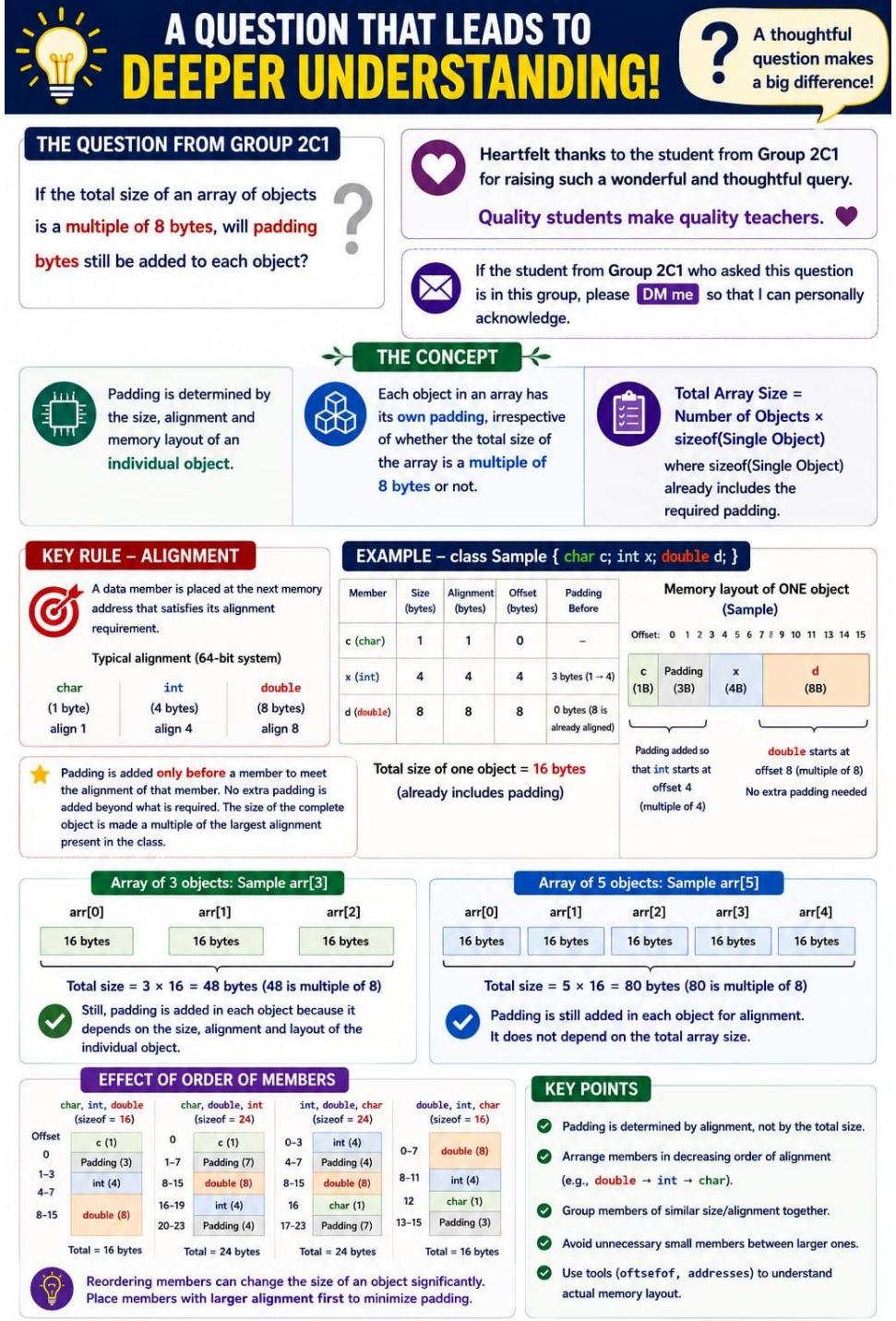
# Nested Classes Example

```
#include<iostream>
using namespace std;
class Enclosing                          /* start of Enclosing class declaration */
{
      int x;
      class Nested
      {                                  /* start of Nested class declaration */
            int y;
      };                                  // declaration Nested class ends here
      void EnclosingFun(Nested n)
      {
            cout<<n.y;       // Compiler Error: y is private in
Nested    }
};                                       // declaration Enclosing class ends here
int main(){}
```


THAPAR INSTITUTE
OF ENGINEERING & TECHNOLOGY
(Deemed to be University)

# Week 3

# Memory Layout






# Contents

- Type Casting
- Object Call by value and call by reference
- Nested Member Functions
- Friend Functions
- Constructors and Destructors

# Type Casting

- Two types
  - Implicit—also called "Automatic"
    - Done for you, automatically
      17 / 5.5
      This expression causes an "implicit type cast" to take place, casting the 17 → 17.0
  - Explicit type conversion
    - Programmer specifies conversion with cast operator
      (double)17 / 5.5
      Same expression as above, using explicit cast
      (double) myInt / myDouble
      More typical use; cast operator on variable

# Program illustrating Typecasting

```
#include <iostream>
using namespace std;
int main()
{
        int a=2,b=5,c,d;
        float e,f=5.0,g,h; c=0
        c=a/f;
        f=a/5.5;
    e=a/b;
    h= (float)a/(float)b; // C notation
    g=float(a)/float(b); //C++ notation
    cout<<"c="<<c<<endl<<"f="<<f<<endl
    cout<<"e="<<e<<endl<<"g="<<g<<endl<<"h="<<h;
}
```

Output

```
c=0
f=0.363636
e=0
g=0.4
h=0.4
```




# Call by value and call by Reference

- Object can be passed as a function argument, like any other data type to member functions as well as non-member functions.
- Pass-by-value
- Pass-by-reference

# Call by value(object)

```
#include <iostream>
using namespace std;
class Convert{
    public :
        int i;
        void increment(Convert obj){
            obj.i = obj.i*2;
            cout << "Value of i in member function : " << obj.i;
        }
};
int main(){
    Convert obj1;
    obj1.i = 3;
    obj1.increment(obj1);
    cout << "\nValue of i in main : " << obj1.i << "\n";
    return 0;
}
```

Output
Value of i in member function : 6
Value of i in main : 3

# Call by Reference

```
#include <iostream>
using namespace std;
class Convert
{   public :
        int i;
        void increment(Convert &obj){
            obj.i = obj.i*2;
            cout << "Value of i in member function : " <<
obj.i;
        }
};
int main (){
    Convert obj1;
    obj1.i=3;
    obj1.increment(obj1);
```

Output:
Value of i in member function : 6
Value of i in main : 6

# OBJECTS + FUNCTIONS

**Pass them • Return them • Store them**

## ① PASS OBJECT

```
void display(Student s) {
    cout << s.marks;
}

Student s1;
display(s1);
```

**Object → Function**

Why? Process an object's data inside another function.

## ② RETURN OBJECT

```
Student createStudent() {
    Student s;
    s.marks = 85;
    return s;
}

Student s1 = createStudent();
```

**Function → Object**

Why? Let a function construct and deliver a complete object.

## ③ ARRAY OF OBJECTS

```
Student s[3];

s[0].display();
s[1].display();
s[2].display();
```

**Many Objects → Array**

Why? Manage a collection of similar objects efficiently.

**⭐ WHY IS THIS IMPORTANT?**

**Reusability • Modular design • Cleaner programs • Easy handling of complex data • Natural foundation for OOP**

# Nesting of Member Functions

- A member function calling another member functions  is known as nesting of member functions.
- A member function of a class can be called only by an object of that class using a dot operator.
- However, there is an exception to this. A member function can be called by using its name inside another member function of the same class.
- Private member functions can also be accessed by using nesting of member functions.

# Nesting of Member Functions

```
#include <iostream>
using namespace std;
class emp{
    float basic;
public:
    void display(){
        cout<<basic;
    }
    void takeData(){
        cin>>basic;
        // A member function of a class can be called by
        // another member function of the same class
        // takeData() is calling diaplay()
        display();
    }
};
int main(){
    emp e1;
    e1.takeData();
}
```

# Exercise (10 minutes)

- Write a program in C++ to create a class Data having member functions get_data() to input the numbers in an array, largest() to find the largest number in the array and display () to print the largest number. Member function largest () and all the data variables should be  private.

```cpp
#include <iostream>
using namespace std;
class Data{
    int num[20];
    int n;
    int largest(void);
public:
    void get_data(void);
    void display(void);
};
void Data::get_data(void){
    cout<<"Enter the total numbers(n):"<<endl;
    cin>>n;
    cout<<"Enter the number:"<<endl;
    for(int i=0;i<n;i++){
        cout<<"Enter the number"<<i+1<<";"; cin>>num[i];
    }
}
```

```cpp
int Data::largest(void){
    int max;
    max = num[0];
    for(int i=1;i<n;i++){
        if(max<num[i])
            max=num[i];
        }
    return max;
}
void Data::display(void){
    cout<<"The largest number:"<<largest()<<endl;
}//nesting
int main(){
    Data num;
    num.get_data();
    num.display();
}
```

# Friend Function

- A function that can access the private and protected members of a class to which it is a friend.
- It is not in the scope of a class to which it is a friend.
- It cannot be called using the object of a class to which it is a friend.
  - Invoked like a normal function.
- Has to be declared either in the public or the private section within a class (to which it is a friend) preceded with a keyword ***friend***.

# Contd…

- Defined elsewhere in the program like a normal C++ function.
- Can be a simple function or a member function of some other class.
- Usually, it has the objects as arguments.
- Best suited in operator overloading.

# Example

```
#include <iostream>
using namespace std;
class myclass{
    int a, b;
public:
    friend int sum(myclass x);
    void set_ab(int i, int j);
};
void myclass::set_ab(int i, int j){
    a = i;
    b = j;
}
int sum(myclass x){
    return x.a + x.b;
}
int main(){
    myclass n;
    n.set_ab(3, 4);
```

# Friend of more than one class

```
#include <iostream>
using namespace std;
const int IDLE = 0;
const int INUSE = 1;
class C2; // forward declaration
class C1 {
    int status;
public:
    void set_status(int state);
    friend int idle(C1 a, C2 b);
};
class C2 {
    int status;
public:
    void set_status(int state);
```

```
void C1::set_status(int state){
    status = state;
}
void C2::set_status(int state){
    status = state;
}
int idle(C1 a, C2 b){
    if(a.status || b.status)
        return 0;
    else
        return 1;
}
```

# Contd…

```
int main(){
    C1 x;
    C2 y;
    x.set_status(IDLE);
    y.set_status(IDLE);
    if(idle(x, y))
        cout << "Screen can be used.\n";
    else
        cout << "In use.\n";
        x.set_status(INUSE);
    if(idle(x, y))
        cout << "Screen can be used.\n";
    else
        cout << "In use.\n";
    return 0;
```

Output

Screen can be used.

In use

# Class member as a friend function

```
1.  #include <iostream>
2.  using namespace std;
3.  const int IDLE = 0;
4.  const int INUSE = 1;
5.  class C2; // forward declaration
6.  class C1 {
7.        int status;
8.     public:
9.        void set_status(int state);
10.       int idle(C2 b);
11. };
12. class C2 {
13.       int status;
14.    public:
15.       void set_status(int state);
16.       friend int C1::idle(C2 b);
17. };
18. void C1::set_status(int
    state)
19. {   status = state;}
20. void C2::set_status(int
    state)
21. {   status = state;}
22. int C1::idle(C2 b)
23. {   if(status || b.status)
24.                return 0;
25.       else
26.                return 1;        }
```

# Contd...

```
27.  int main()
28.  {    C1 x;
29.       C2 y;
30.       x.set_status(IDLE);
31.       y.set_status(IDLE);
32.       if(x.idle(y))
33.                 cout << "Screen can be used.\n";
34.       else
35.                 cout << "In use.\n";
36.       x.set_status(INUSE);
37.       if(x.idle(y))
38.                 cout << "Screen can be used.\n";
39.       else
40.                 cout << "In use.\n";
41.       return 0;
```

Output

```
Screen can be used.
In use.
```

# Friend Class

- A class can also be made a friend of another class.
- Example:

```
class A
{   ......
};
```

```
class B
{ .....
  friend class
   A;
 .....  };
```

- All the member functions of a friend **class A** become friend of **class B**.
- Any member function of **class A** can access the private data of **class B**.
- But, member functions of **class B** cannot access the private data of **class A**.

# Example

```
#include <iostream>
using namespace std;
class TwoValues {
    int a, b;
    public:
    TwoValues(int i, int j) { a = i; b = j;
}
    friend class Min;
};
class Min {
    public:
    int min(TwoValues x);
};
int Min::min(TwoValues x){
    return x.a < x.b ? x.a : x.b;
}
```

Output
10

```
int main(){
    TwoValues ob(10, 20);
    Min m;
    cout << m.min(ob);
    return 0;
}
```


दृश्य एवं संकेत सूचना प्रसंसकरण समूह
Visual Signal and Information Processing Group
TIET


ti
THAPAR INSTITUTE
OF ENGINEERING & TECHNOLOGY
(Deemed to be University)

# Thank You!