Notes:- C++

→Lecture 1(d-2-7-25):-

Introduction –

- C++ language is developed in 1985 by Bjarne stropstrup.

- C++ is a cross -platform language that can be used to create high performance application.

--currently in gui,os and embeeded system.

--advantage:-resuablity of code.

aspect	C	C++
Developer	C was developed by Dennis Ritchie between the year 1969 and 1973 at AT&T Bell Labs.	C++ was developed by Bjarne Stroustrup in 1985 first comersiyerly release.
OOPs Support	C does not support polymorphism, encapsulation, and inheritance which means that C does not support object-oriented programming.	C++ supports polymorphism, encapsulation, and inheritance because it is an object-oriented programming language.
Subset/Superset	C is (mostly) a subset of C++.	C++ is (mostly) a superset of C.
Keywords	Number of keywords in C:	Number of keywords in C++:

	- C90: 32 - C99: 37 - C11: 44 - C23: 59	- C++98: 63 - C++11: 73 - C++17: 73 - C++20: 81
Programming Paradigm	For the development of code, C supports procedural programming.	C++ is known as hybrid language because C++ supports both procedural and object-oriented programming paradigms.
Encapsulation	Data and functions are separated in C because it is a procedural programming language.	Data and functions are encapsulated together in form of an object in C++.
Data Hiding	C does not support data hiding.	Data is hidden by the Encapsulation to ensure that data structures and operators are used as intended.
Focus of Language	C is a function driven language because C is a procedural programming language.	C++ is an object driven language because it is an object-oriented programming.
Overloading	Function and operator overloading is not supported in C.	Function and operator overloading is supported by C++.

Function Inside Structures	Functions in C are not defined inside structures.	Functions can be used inside a structure in C++.
Namespaces	Namespace features are not present inside the C.	Namespace is used by C++, which avoid name collisions.
Standard I/O	Standard IO header is stdio.h and uses scanf() and printf() functions are used for input/output in C.	Standard IO header is iostream.h and uses cin and cout are used for input/output in C++.
References	Reference variables are not supported by C.	Reference variables are supported by C++.
Virtual Functions	Virtual and friend functions are not supported by C.	Virtual and friend functions are supported by C++.
Inheritance	C does not support inheritance.	C++ supports inheritance.
Dynamic Memory	C provides malloc() and calloc() functions for dynamic memory allocation, and free() for memory de-allocation.	C++ provides new operator for memory allocation and delete operator for memory de-allocation.

Exception Handling	Direct support for exception handling is not supported by C.	Exception handling is supported by C++.
Access Modifiers	C structures don't have access modifiers.	C ++ structures have access modifiers.
Type Checking	There is no strict type checking in C programming language.	Strict type checking in done in C++. So many programs that run well in C compiler will result in many warnings and errors under C++ compiler.
Type Punning with Unions	Type punning with unions is allows (C99 and later)	Type punning with unions is undefined behavior (except in very specific circumstances)
Named Initializers	Named initializers may appear out of order	Named initializers must match the data layout of the struct
Extension	File extension is ".c"	File extension is ".cpp" or ".c++" or ".cc" or ".cxx"
Generic Programming	Meta-programming using macros and _Generic()	Meta-programming using templates (macros

		are still supported but discouraged)

→

C++ Language:

Lecture -1 :-date(2-7-25)

→**class=**

data + functions

Data member AND MEMBER FUNCTION

Blueprint

→**object** –

Real entitiey

Access data and function of a class

Object is type of class

Int a=20;

Ex.class person{};

→**Encapsulation**-data member+member function in single unit(hiding data private)

→**Abstraction** -showing only essential detail

→**Polymorphism** -more than one form

Two type of polymorphism

→Compile time/static binding/method overloading

→Runtime/dynamic binding/method overriding

→**Inheritance**-property on eclass can use in another class.

-reusability of code

-parent -child

```
Clas person{
Int contact_number,age;
Void getdata(){
}};
```

```
Cout<<"hello world";
Console output;
<<insertion operator
Std==namespace
Cin>>
Console input;
>>Extraction operator;
```

→**lecture** -2:- (d-3-7-25)

Datatype

Primitive

Int

Float

Char

Access modifier-

By default in c++ all class member and function are private.

Public

Private

protected

→lecture -3(d-4-7-2025)

Operator

-bitwise

10=01010

20=10100

&= 00000(and)

|= 11110(or)

^= 11110(Xor)

~= 10101(not)

<<(left shift)=10100=20

>>(right shift)=01010=10;

→**function in** c++:-

Default value :-

In only posiblen in c++not in c.

→and one more thing flow is right to left for default argument.

- Function overloading

→String in C++:

- String is sequence of charcter
- In c++ we can use string using inbuild library <string>
- String is datatype in c++ so no need to take char array for string like c
- Here some different function
- Variable.length()
- Variable.replace(pos,len,”word”);
- Variable.size()//same as length return length
- Variable.append(word)//joint word at last
- Variable.insert(pos,”word”);//joint at particular location
- Variable.substr(pos,poe)//find string in between pos and poe.

→Encapsulation :-

- Wrapping up to data member and member function into single unit
- Also hide sensitive info.
- Geter and setter function used for access privete data

→Constructure:-

- same name as class name
- automatically called when object created
- no return type
- initialize class
- constructur overloading
- types of constructure
- 1.default
- 2.parameterized

- 3.copy constructure
- ---

Lecture-4(d-5-7-25)

→Inheritance

- Property of one class inherited by or use by another class.
- Data access of another class
- Parent chid
- reusablity
- Types of inheritance
- 1.single inheritsnce
- 2.multiple inheritsnce
- 3.multilevel inheritance
- 4.hierarchical inheritance
- 5.hybrid inheritance.

1.single inheritance

```
Class a{

};

Class b:public b{

};
```

2.multilevel

```
Class grand parent

{}

Class parent:public grandparent{}

Class child:parent{}
```

3.multiple

```
Class a{}
```

Class b{}

Class c:a,b{}

4.hierarchy

Class a{}

Class b:a{}

Class c:a{}

Hybrid:

Class a

:: scope resolution operator

Lecture-5 (d-7-7-2025)

- Test

Lecture -6 (d-8-7-2025)

- Polymorphism
- **Polymorphism** means **"many forms"** — it allows **one name** (like a function or operator) to behave **differently based on the context**.
-
- 🔥 **Simple Definition:**
- Polymorphism allows the **same function name** or operator to work **differently** depending on the **object** or **input**.
-
- 🧠 **Types of Polymorphism in C++:**

Type	Also Known As	Example
Compile-time Polymorphism	Static binding,early binding	Function Overloading, Operator Overloading,
Runtime Polymorphism	Dynamic binding,late binding	Virtual Functions (with inheritance),method overrideng

-
-

❖ Compile time polymorphism:

→method overloading

- Same method name but different type and number of argument

→Operator overloading

- **Operator Overloading** means giving **special meaning** to a **built-in operator** (like +, –, *, etc.) when used with **user-defined data types** (like classes and objects).
-
- 🔥 **Simple Definition:**
- **Operator Overloading** allows operators to **work with objects** just like they work with basic data types.

→Lecture -7(9-7-2025)

- →Operator overloading
- d3=d1+d2;
- D1 operator+(d2){
- //means first argument as return and second under bracket for work
- }
- Binary operatoar(+,-,*,/)
- Working with more than one operand
- Unary operator(++,--)
- Working with only one operand
- This keyword or this pointer is used to reference current object.

→lecture -8 (10-7-25)

Lecture-8(10-7-25)

→method overriding

- Method have same name same parameter and same type
- Used in inheritance
- Must write virtual keyword in base class to overrid or change data for derived class in same method.

→Scope resolution operator(::)-

- used to global variable and function .
- used to reperent same name function or method to clarify which method of which class when both class have same name method using ::.

- **Situation**	- **Scope resolution needed?**	- **Why**
- Defining class functions outside class	- ✅	- To associate with the class
- Accessing global variable when local exists	- ✅	- To clarify scope
- Using namespaces	- ✅	- To avoid collisions
- Accessing static class members	- ✅	- They belong to class, not object

---

Lecture-9(11-7-2025)

→Abtraction- showing essential details and hiding detailing

- An **abstract class** is a class that **cannot be instantiated** (i.e., you cannot create objects of it) and is used as a **base class** for other

classes. It is used to **define a common interface or structure** that derived (child) classes must follow.

- Also used in inheritance
-

Key Points:

- **Contains at least one pure virtual function**.
- Used to **enforce a contract** for subclasses.
- Child classes **must override** the pure virtual functions.
- You **cannot create objects** of an abstract class.

Why Use Abstract Classes?

- To **enforce a standard interface** for all derived classes.
- To provide **polymorphic behavior**.
- To achieve **partial implementation**, where base class provides some code and expects child classes to implement the rest.
-

- Data hiding by making them private(c_no,salary)
- Public(get(),display()

→Pure virtual function:-

- A **pure virtual function** is a function in a base class that **has no body** and **must be overridden** by derived classes.
- It is used to create **abstract classes**, which define a **common interface** for all derived classes.

Ex:

Virtual void msg()=0;

Key Points:

- A class with at least one pure virtual function is an **abstract class**.
- You **cannot create objects** of an abstract class.
- Derived classes **must override** the pure virtual function(s).

Why Use Pure Virtual Functions?

- To define a **standard interface** (set of rules) for child classes.
- To support **runtime polymorphism**.
- To make sure every derived class **must implement** its own version of the function.
- **Difference Between Pure Virtual Function and Simple Virtual Function in C++**

Feature	🧩 Simple Virtual Function	🎯 Pure Virtual Function
Definition	A function with a body in base class	A function with no body in base class (`= 0`)
Syntax	`virtual void show() {}`	`virtual void show() = 0;`
Must Override in Derived?	❌ Not necessary (optional override)	✅ Must be overridden in derived class
Creates Abstract Class?	❌ No	✅ Yes
Object Creation	✅ You can create objects of the class	❌ You **cannot** create objects of abstract class
Purpose	Allows **optional** overriding and polymorphism	Enforces **mandatory** overriding (interface-style)

→1.if class contain one or more pure virtual function then that class is abstract class.

→2. Can not create object of abstract class

→3.must have override pure virtual function in derived class.

→4.virtual function it have body{}

→5.Pure virtual function not have body but =0;

→difference between abstract and concreat class:

- 1.Object- you can not create object of abstract class
- While you can create object of concreat class.
- 2.methods- Abstact class must contain one pure virtual function
- Abstarct class have normal function also.
- While concreat class not have any pure virtual function.
-

Feature	Abstract Class	Concrete Class
Can create objects?	❌ No	✅ Yes
Pure virtual function?	✅ At least one	❌ None
Purpose	Provides a structure/interface	Provides full implementation
Usage	Used as base class only	Can be used directly

→Difference between Virtual and normal function method : -

- Virtual function must be override while normal may or may not be overridden

- A **virtual function** is a member function in a base class that you **expect to override** in derived (child) classes. It supports **runtime polymorphism** (i.e., the correct function is called **based on the object type**, not pointer type).

- **Feature**	- 🟢 **Normal Function**	- 🔵 **Virtual Function**
- **Definition**	- Regular member function	- Declared with virtual keyword
- **Overriding in Derived Class**	- Possible, but resolved at **compile time**	- Overriding is resolved at **runtime**
- **Polymorphism Support**	- ❌ No **runtime polymorphism**	- ✅ Supports **runtime polymorphism**

- **Function Call Based On**	- Type of **pointer/reference**	- Type of **actual object (instance)**
- **Keyword Used**	- None	- virtual
- **Use Case**	- General behavior	- To allow different behavior in derived classes

→

→**Lecture**-10(12-7-25):-

→inline function

All function in class definition are also inline .

Simple function there is no complexity

Its also give code of function at function call no need every time to jump in class to main or main to class(my understanding)

No loop

No recursive

→friend function,class

A **friend function** is a **non-member function** that is allowed to access **private** and **protected** members of a class

Access the private data of classes using simple function

- It is not a part or member of class but also must be declare in class ;
- Friend keyword
- Ex;
- Friend void display1(A);
- Also pass class and its object as argument.
-

→friend vs normal function

-private data access by friend while normal function not access

Member not a friend while normal also

- Calling
- Parameter must class object
- Must be declare first in class friend function
- While normal no need any declaration in class;

→ **Lecture** -11(14-7-2025)

- Friend function /friend class
- Main goal is private or protected data access.

→most importanant

→static data -concept

1. Static Data Member

A **static data member** is shared by **all objects** of a class instead of having separate copies for each object.

✳️ **Characteristics:**

- Declared with the static keyword **inside** the class.
- Defined **outside** the class using ClassName::member.
- Memory is allocated **only once** for the static member, no matter how many objects are created.
- Can be accessed using:
  - object (obj.member) ✅
  - class name (ClassName::member) ✅

- Static data initialize only ones.
- Its not belong to object its belong to class
- Its access by return type class name::any method or data initialize
- Its also when you required data only once time initialize.
- For memory management purpose its also use

→static method:-

2. Static Member Function

A **static member function** belongs to the class, **not to any object**.

✳️ **Characteristics:**

- Declared using the static keyword inside the class.
- Can access **only static data members** or call **other static functions**.
- Cannot access this pointer (because it doesn't belong to any object).
- Can be called using:
    - class name (ClassName::function()) ✅
    - object (obj.function())

- Static method can not use non static data.
- Static method also use static data.
- Todays task.
- Employee s data are multiple but company name is same

Lecture-12(d-15-7-2015)

- Malloc and calloc(dynamic memory allocation run time memory allocation

- Malloc take only one parameter it will be size ,by defaut garbage values initialize
- Calloc takes two parameters ,by defaut 0 initialize
- Both function return * void pointer.
-
- Generally used in array
- **malloc() (Memory Allocation)**
- 🔹 **Purpose:**
- Allocates a **block of memory** of given size in **bytes**.
- Does **not initialize** the memory (it may contain **garbage values**).
- 🔸 **Syntax:**
- c
- Copy code
- void* malloc(size_t size);
- 🔸 **Example:**
- c
- Copy code
- int *ptr = (int*)malloc(5 * sizeof(int)); // allocates memory for 5 integers
- 🔸 **Important Notes:**
- Returns a void* pointer → needs to be typecast.
- If allocation fails, returns NULL.
- Memory content is **not zeroed**.
- ---
- ✅ **calloc() (Contiguous Allocation)**
- 🔹 **Purpose:**
- Allocates **multiple blocks** of memory and **initializes all to 0**.
- Used for arrays where initial zeroing is needed.
- 🔸 **Syntax:**
- c

- Copy code
- void* calloc(size_t num, size_t size);
- 🔸 **Example:**
- c
- Copy code
- int *ptr = (int*)calloc(5, sizeof(int)); // allocates memory for 5 integers, all initialized to 0
- ---
- 🔄 **malloc() vs calloc()**

Feature	malloc()	calloc()
Initialization	No (contains garbage)	Yes (initializes to 0)
Parameters	1 → total size in bytes	2 → number of blocks and size of each block
Speed	Slightly faster	Slightly slower (due to initialization)

- ---
- 🧹 **Freeing Memory**
- Both malloc() and calloc() use heap memory. You must **free** it manually:
- c
- Copy code
- free(ptr); // to prevent memo
-

→destructor:-

A **destructor** is a **special member function** in C++ that is **automatically called** when an object **goes out of scope** or is **explicitly deleted**. It is

used to **free resources** that the object may have acquired during its lifetime.

What is a Destructor?

A **destructor** is a special member function in C++ that is automatically called **when an object goes out of scope** or is **explicitly deleted**.
Its purpose is to **release resources** like memory, file handles, or network connections.

- **Key Features:**

Feature	Description
Auto-invoked	Called automatically when object is destroyed
One per class	No overloading allowed
No parameters	Cannot take arguments
Used for	Cleanup (memory, files, etc.)

- **Syntax of Destructor:**

~ClassName();

- Same name as the class, **preceded by a tilde ~**
- No return type
- No parameters
- Only one destructor per class (not overloaded)

~Class name(){}

→free function

- Its memory deallocation function in c used with stdlib library
- It also used when with malloc and calloc only.

Free();

→**file handling**:

File handling in C++ allows programs to **read from** and **write to files** stored on disk using file streams provided by the **<fstream>** header.

- **1. Header File Required**

cpp

Copy code

#include <fstream>

This provides three main classes:

Class	Purpose
ifstream	Input file stream (read)
ofstream	Output file stream (write)
fstream	File stream for both reading and writing

- **2. Opening a File**
- **Using constructor:**

cpp

Copy code

```
ofstream fout("file.txt");   // write
ifstream fin("file.txt");    // read
```

- **Using .open():**

cpp

Copy code

```
fstream file;
file.open("file.txt", ios::out);  // write mode
```

- **3. File Modes (ios flags)**

Mode	Description
ios::in	Open file for reading
ios::out	Open file for writing
ios::app	Append to end of file
ios::binary	Open file in binary mode
ios::trunc	Delete contents if file exists
ios::ate	Go to end of file after opening

You can combine modes using |:

cpp

Copy code

```
file.open("file.txt", ios::in | ios::out);
```

🔹 4. Basic File Operations

✅ Writing to File:

cpp

Copy code

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

int main() {
    ofstream fout("data.txt");  // open for writing
    fout << "Hello, world!\n";
    fout << "File handling in C++.";
    fout.close();  // always close the file
    return 0;
}
```

✅ Reading from File:

cpp

Copy code

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

int main() {
    ifstream fin("data.txt");  // open for reading
```

```
    string line;

    while (getline(fin, line)) {
        cout << line << endl;
    }

    fin.close();
    return 0;
}
```

- **5. Check if File is Opened**

cpp

Copy code

```
ifstream fin("data.txt");
if (!fin) {
    cout << "File could not be opened.";
}
```

- **6. Closing a File**

cpp

Copy code

```
fout.close();
fin.close();
```

- Always close files to free resources and save data properly.

📝 Summary Table

Operation	Object Used	Example
Write	ofstream	ofstream fout("a.txt");
Read	ifstream	ifstream fin("a.txt");
Both	fstream	`fstream f("a.txt", ios::in

Read,write,append;

Open read/write/close;

- Fstream header file also include
- Ifstream-reading only
- Ofstream-write only
- Fstream-read/write both
- Cout
- Cin

→lecture-13(16-7-25)

- file handling:
- task
- menu driven
- 1.to read-which file?
- Press 1 for Read by line/
- Press 2 read by word
- 2.to write-which file
- 3 to append ios::app

→Lecture -14(17-7-25)

- File
- **1. Functions to Check & Move File Pointer**
- 🔹 **For Input (`ifstream`, `fstream`):**

Function	Purpose
`tellg()`	Get current **read** position (get pointer)
`seekg(pos)`	Move the **read** pointer to `pos`

- 🔹 **For Output (`ofstream`, `fstream`):**

Function	Purpose
`tellp()`	Get current **write** position (put pointer)
`seekp(pos)`	Move the **write** pointer to `pos`

- ---
- ✅ **2. Example: Read from a Specific Position**

```
- cpp
- Copy code
- #include <iostream>
- #include <fstream>
- using namespace std;
-
- int main() {
-     fstream file;
-     file.open("example.txt", ios::in);  // Open for reading
-
-     if (!file) {
-         cout << "Error opening file!";
-         return 1;
-     }
-
-     file.seekg(6);  // Move to position 6 (7th character)
-     char ch;
```

```
    file.get(ch);   // Read a single character
    cout << "Character at position 6 is: " << ch << endl;

    file.close();
    return 0;
}
```

✅ 3. Example: Write at a Specific Position

```
cpp
Copy code
#include <iostream>
#include <fstream>
using namespace std;

int main() {
    fstream file;
    file.open("example.txt", ios::in | ios::out);  // Open for read/write

    if (!file) {
        cout << "Error opening file!";
        return 1;
    }

    file.seekp(5);  // Move write pointer to position 5
    file << "X";    // Overwrite 6th character with 'X'

    file.close();
    return 0;
}
```

✅ 4. Check Current Pointer Position

```
cpp
Copy code
cout << "Current read position: " << file.tellg() << endl;
cout << "Current write position: " << file.tellp() << endl;
```

🔄 Offset Movement Using `seekg()` / `seekp()`

```
cpp
Copy code
file.seekg(5, ios::beg);  // from beginning
file.seekg(-2, ios::end); // 2 chars before end
file.seekg(3, ios::cur);  // 3 chars from current
```

Same applies for `seekp()`.

📝 Summary:

Function	Used With	Purpose
`tellg()`	ifstream	Get current read position
`seekg(pos)`	ifstream	Set current read position
`tellp()`	ofstream	Get current write position
`seekp(pos)`	ofstream	Set current write position

-
- Position -current ,ask/put
- File.tellg() function by default
- File.seekg(9) take position
- Fstream
- Escape sequence=\n,\t,\".
- Ios::out=write
- Ios::in=read
- Ios::app=append