Day 3

Intro C++

- 1979
- Bjarne Stroustrups
- Bell Lab
- C with classes
- 89 -> C++ -> ANSI -> ISO group
- Standardization of C++
    - C++89
    - C++98
    - C++03
    - C++11
    - C++14
    - C++17
    - C++20
    - C++23
    - C++26

1. more control on system resources
2. memory management
3. fast program
4. high performance

Data Types

- Basic data types:
    - int, char, double, float, void, bool, wchar_t
- Derived data types:
    - Array, Pointer, Function
- User Defined Data Types:
    - Structure, Union, Enum, Class

Type Modifiers

- signed, unsigned, short, long

Specifiers/Qualifiers

- const, volatil

Structure in C++

- Limitation of C Structure
    - The structure members are by default public
    - Inside c structure we cannot have function, constructors, destructors.
    - Access specifiers are not there.
    - Implicitely no need to write struct

```cpp
#include <cstdio>

struct Student{
private:
    int rollNo;
    char name[20];

public:
//struct member function
    void setData(){
            printf("Enter Roll No: ");
            fflush(stdout);
            scanf("%d", &rollNo);
            printf("Enter Name: ");
            fflush(stdout);
            scanf("%s", name);
    }

    void printData(){
        printf("RollNo : %d \n", rollNo);
        printf("Name : %s \n", name);
    }
};

int main(){
    Student s1;

    s1.setData();
    s1.printData();

    return 0;
}
```

Access specifiers

- public - Members can be accessed from anywhere.
- private - Members can be accessed only within the same class.
- protected - Members can be accessed within the same class and by derived class.

Class and Object

Class

- user defined data type
- Blueprint/Template
- Data members + Member functions

Object

- instance of class

- Difference between struct and class
- struct -> By-default public
- class -> By-default private

```
#include <cstdio>
#include <cstring> // for strcpy
struct Student1{
    //public -> by-default
    int rollNo;
    char name[20];
};
class Student{
    //private -> by-default
    int rollNo;
    char name[20];
};

int main(){
    Student s1;
    s1.rollNo = 10;
//  s1.name = "shil";
//  strcpy(s1.name, "shil"); // C style string copy

    printf("Roll No : %d", s1.rollNo);
//  printf("Name : %s", s1.name);

    return 0;
}
```

Getters & Setters

Getters

- To get the data of private variable, provides read access

Setters

- To set the data for private variable, provides write access

```
#include <iostream>
#include <string>

class Student{
    int rollNo;
    std::string name;

public:
    // Getter for rollNo
    int getRollNo(){
```

```
        return rollNo;
    }
    // Getter for name
    std::string getName(){
        return name;
    }
    // Setter for rollNo
    void setRollNo(int rollNo){
        this->rollNo = rollNo;
    }
    // Setter for name
    void setName(std::string name){
        this->name = name;
    }
};

int main(){
    using namespace std;
    Student s1;  // this is on stack
                 // new -> it will be on heap
    s1.setRollNo(10);
    s1.setName("shil");

    cout << "Roll No : " << s1.getRollNo() << endl;
    cout << "Name : " << s1.getName() << endl;

    return 0;
}
```

Empty class

- Size of empty class is 1 byte
- Why 1 byte?
    - To provide unique address to each object of the class

```
#include <iostream>

// Employee class with data member
class Employee{
    int id;
};
// Empty Student Class
class Student{

};
// Inheritance Example
class A{

};
class B:A{
```

```
};
int main(){
    printf("%u", sizeof(Student));
    printf("%u", sizeof(Employee));
    printf("%u", sizeof(B));
    return 0;
}
```

Data Members

- Attributes, instance variable
    - const - Something which we cannot change/ read-only variables
    - static - One things which shared among all the object of your class

Member Fuctions

- const
- static
- virtual

Stream concept

- Flow of data from one source to another
- Source -> Destination
- Input stream -> getting data from source to program
- Output stream -> sending data from program to destination
- Standard stream objects associated with console.
- cin, cout, cerr and clog objects Example:
    - user (FE) -> BE -> DB
    - file -> data -> HDD
- console -> consist of Monitor + Keyboard
- In case c -> for stdout -> printf, for stdin -> scanf, for stderr -> error
- In C++ -> for stdout -> cout for stdin -> cin for stderr -> cerr for log -> clog
- for cout ->

```
typedef basic_ostream& ostream
```

- for cin ->

```
typedef basic_istream& istream
```

cerr and clog ->

```
typedef basic_ostream& ostream
```

- input -> getting data from user from console -> Keyboard
- output -> showing data to user from console -> Monitor

C style

```
 \n -> new line,
 \t -> tab space,
 %.2f -> float with 2 decimal point
```

C++ style

```
 endl -> new line + flush the buffer
 setw(n) -> setw is used to set width of output field to n
 fixed << setprecision(2) -> float with 2 decimal point
 scientific << setprecision(2) -> float with 2 decimal point in scientific
notation
 hex -> hexadecimal
 dec -> decimal
```

- Example

```
#include <iostream>

using namespace std;

class Student{

public:
    int rollNo;
    string name;

    void setData(){

        //c style
//      printf("Enter name: ");
//      scanf("%s", name);
```

```
        //C++ style
        cout << "Enter Roll No: " << endl; //'\n' \t %.2f endl setw, scientific,
hex, dec, etc.
        cin >> rollNo;

        if(rollNo == 0){
            cerr << "Roll No. cannot be zero" << endl;
        }

        clog << "Roll No entered success" << endl;

        cout << "Enter Name :" << endl;
        cin >> name;

        clog << "Name entered success" << endl;
    }


    void printData(){

//      printf("Roll No : %d, Name : %s", rollNo, name);

        cout << "Student Details" <<endl;
        cout << "Roll No : " << rollNo << endl;
        cout << "Name : " << name << endl;
    }
};


int main(){
    Student s1;

    s1.setData();
    s1.printData();

    return 0;
}
```

Header Guard

- To avoid multiple inclusion of header files
- Prevents redefinition errors
- Syntax:

```
#ifndef MACRO_NAME
#define MACRO_NAME

//header file code

#endif /* MACRO_NAME */
```

- Example:

```
#ifndef STUDENT_H_   // header guard -> STUDENT_H_ it is a macro -> if it is already defined then ignore it
#define STUDENT_H_  // if it not defined -> define the macro

//#pragma once  // no standardization -> not recommended -> but industry project they use it, most of the compilers allows it

//code
#include <iostream>

class Student{
public:
    int rollNo;
    std::string name;
    void setData();
    void printData();
};
#endif /* STUDENT_H_ */  // endif -> this header section or macro definition section ends
```

- pragma once
  - Non-standard but widely supported by modern compilers
  - Simpler syntax
  - Prevents multiple inclusions

#include < > vs " "

#include < >

- Used for standard library headers
- Compiler searches in system directories

#include " "

- Used for user-defined headers
- Compiler searches in the current directory first, then system directories

For Multi-file project

```
- main.cpp
- student.h
- student.cpp
```

- Directory Structure

```
    |-- src
        |-- main.cpp
        |-- student
                |-- student.h
                |-- student.cpp
```

- If you want to include student.h in main.cpp then you have to use `#include "student/student.h"`
- `We can say we have acheived modularity in C++ using header files and source files.`
- Example:

```
//not valid
//#include <student.h>  // error: student.h: No such file or directory -> because
it tries to find this in std directory

//src -> main.cpp -> student.h include krna hai -> #include "student/student.h"
//src -> /student
//      - student.h
//      - student.cpp

//import java.util.Array; // in java

//Modularity

#include <iostream>
#include "student.h" // include user defined file

int main(){
    Student s1;
    s1.setData(); // member function call -> error if we do not have prototype
    s1.printData();
    return 0;
}
```

Storage Classes in C++

- This are the classes which are used to define scope of variables and lifetime.

auto

- This is by default it is present in your program, even if you do not write it.

```
 int main(){
    auto int a = 10; // auto keyword is optional
    int b = 20; // by default it is auto
```

```
    return 0;
}
```

register

- The variable you have declared with register keyword it will directly store it in -> CPU register
- faster access -> networking, hardware related
- If register has memory to store it then only it will store -> it will ignore it if it is too big to store
- most compiler are optimized to use it in better manner

```
int main(){
    register int a = 10; // try to store it in CPU register
    return 0;
}
```

static

- local static variable -> inside function -> will retain value b/w function calls
- global static variable
- static data member of class
- static member function of class
- static -> memory will be allocated only once
- one for a class -> variable will be there throughout the class -> can be made global

```
void func(){
    static int count = 0; // local static variable
    count++;
    std::cout << "Count: " << count << std::endl;
}

int main(){
    func(); // Count: 1
    func(); // Count: 2
    func(); // Count: 3
    return 0;
}
```

extern

- It is used to declare a global variable or function in another file.
- It tells the compiler that the variable or function is defined in another file.
- define -> declaration is somewhere else -> but at the place where definition is there give memory
- declare -> main.cpp -> not here
- definition -> student.cpp -> memory here

```cpp
// file1.cpp
#include <iostream>
extern int count; // Declaration of extern variable
void displayCount(){
    std::cout << "Count: " << count << std::endl;
}
```

```cpp
// file2.cpp
#include <iostream>
int count = 10; // Definition of extern variable
int main(){
    displayCount(); // Output: Count: 10
    return 0;
}
```

Lifetime

- The lifetime of a variable is the duration for which the variable exists in memory during the execution of a program.
- Automatic lifetime -> local variable -> automatic
- Dynamic lifetime -> dynamic variable -> dynamic
- Static lifetime -> global variable, static variable -> static

Scopes in C++

- Global scope - outside of all functions and classes
- block scope - inside { }
- local -> inside function
- program -> entire program
- class -> data members, member functions -> accessed using object
- file -> static -> inside file
- function -> parameters, local variables
- namespace -> namespace{ class, variable, function}
- funtion prototype scope -> inside function prototype
- static file you can access it but outside file cannot access it
- non static you can access it outside of the file as well

-Example:

```cpp
#include <iostream>
int globalVar = 100; // Global scope
void func(){
    int localVar = 10; // Local scope
    {
        int blockVar = 20; // Block scope
        std::cout << "Block Var: " << blockVar << std::endl;
```

```
    }
    // std::cout << "Block Var: " << blockVar << std::endl; // Error: blockVar is
out of scope
    std::cout << "Local Var: " << localVar << std::endl;
}
```

Namespace

- It is a declarative region that provides a scope to the identifiers (names of types, functions, variables, etc) inside it.
- helps in avoiding name collisions in large projects
- std namespace
- user defined namespace

```
//student.h
#include <iostream>
namespace my{
    int val = 25; //declaration // different
}

//main.cpp
#include <iostream>
#include "student.h" // int val is defined here

using namespace std; // std namespace

//int val = 10; //redefinition of 'int val'

//namespace -> keyword

//namespace hello{
//
//}

//namespace _nameForNamespcae{}
//c++ library -> my

//namespace my{
//  int val;
//}

// int val;

//namespace frnd{
//  int val;
//}

//c++ library -> friend
// int val;
```

```
int val = 10; // different

int main(){

    int val = 100;

    cout << val << endl;
    cout << ::val << endl;
    cout << my::val << endl;
    return 0;
}
```

Tomorrow Topics:

- Function Overloading
- Name Mangling
- extern “C”
- Enum
- Default arguments
- this pointer