Day 6 (06-12-2025)

const member function

- We cannot change the values in this function
- it is used to make sure that the data members are not changed by mistake.
- it is used in getter functions.

```
class Student{
public:
    int rollNo;  //data member
    string name;

// const Student *const this        // cannot change the data as well-> cannot
point to another object
    int getRollNo() const {         //getter function
//      rollNo = 20;                // error: assignment of member
'Student::rollNo' in read-only object
        return rollNo;
    }

//  Student *const this         // no change in address -> cannot point to another
object
    const void printRecord(){           //it should only print the record
        this->rollNo = 20;
        cout << rollNo << endl;
    } // this function will return a const value
};

int main() {
    int val = 10;

    Student s1;

    s1.rollNo = val;

//  printRecord(&s1);
    s1.printRecord();

    return 0;
}
```

mutable member

- It is used when we want to change the value of variable in const member function.
- it is used in logging, caching etc.

```cpp
//typedef
typedef class StudentAttendanceReport{
public:
    int rollNo;
    string name;
    //mutable keyword
    mutable int attendanceCount = 0; // to change this variable in const member
function

    const StudentAttendanceReport& joined_for_class() const {
        cout << "Roll No: " << rollNo << " "  << "Name: " << name << " " << "is
joined." << endl;
        return *this;
    }

    //const member function
    const StudentAttendanceReport& camera_on() const {
        attendanceCount++;
        cout << rollNo << " cam ON." << endl;
        return *this;
    }

    int getAttedanceCount(){
        return attendanceCount;
    }

} SAR; //changed with typedef

int main() {
    SAR s1;

    s1.rollNo = 100;
    s1.name = "shil";

    s1.joined_for_class().camera_on();      // function chaining -> *this

    cout << "Attendace Count: " << s1.getAttedanceCount() << endl; // attendance
count
    return 0;
}
```

Aggregate initialization

- To initialize the object directly from main function without the need of constructor
- Rules to be kept in Mind while using it:
    - data members -> public, not private
    - explicitely no constructor written
    - Inheritance not allowed ->but c++ 11 -> made it possible -> c++17
    - virtual method not allowed
    - Destructor not allowed

```cpp
//another class inside class
class Course{
public:
    int course_id;
    string course_name;
};

//main class
class Student{

public:
    int rollNo;
    string name;
    Course course; // nested object

//  data members -> public, not private
//  explicitely no constructor written, also can write parameterized constructor
for inheritance c++11/c++17/c++20
//  Inheritance not allowed -> c++ 11 -> made it possible -> c++17
//  virtual method
//  destructor
};

// inheritance example
class A{
public:
    int x;
};

class B : A{  //inheritance
public:
    int y;

    //constructor for B -> to initialize base class member
    // exceptional in aggregate initialization, to resolve the A() init issue
c++11/c++17/c++20
    B(int x, int y) : A{x}, y{y} {
    }
};

int main(){

    int arr[] = {10,20,30,40,50};       //aggregate initialization -> c style
    int arr1[]{10, 20, 30};         //aggregate initialization -> c++ style

//  Student s1 = {100,"shil"};  // aggregate initialization for object-> c style
    Student s1{100,"shil"};     // aggregate initialization for object-> c++ style

    cout << s1.rollNo << " " << s1.name << endl;

    //aggregate initialization for array of objects
    Student s_arr[]{Student{100, "shil"}, Student{101,"raj"}, Student{103,
```

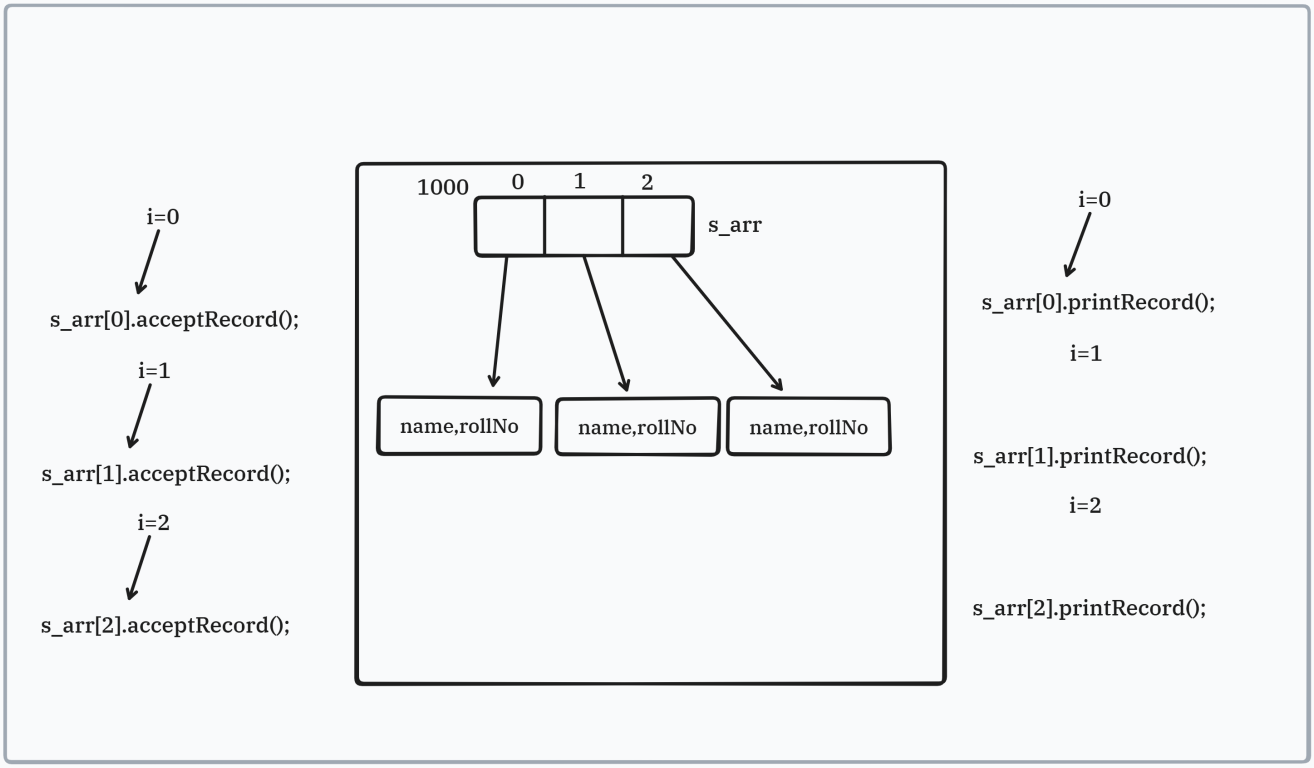
```
"ram"}};  // array of object

    for(int i=0;i<3;i++){
        cout << s_arr[i].rollNo << " " << s_arr[i].name << endl;
    }

    //aggregate initialization with inheritance
//  B b{20}; //c++ 11 -> c++14/c++17 - made it robust
//  B b{{10},20}; // c++17
    B b{10, 20}; // braces elision c++20
    // B b{.y = 20, .x = 10}; // c++20
    cout << b.x << " " << b.y << endl;

    Student s2{145, "shil", {1, "PG-DAC"}};  // c++11
    Student s3{.rollNo = 123, .name = "aditya", {.course_id = 1, .course_name =
"PG-DAC"}}; //c++ 20

    cout << s2.rollNo << " " << s2.name << " " << s2.course.course_id << " "<<
s2.course.course_name <<endl;
    return 0;
}
```

Arrays of objects

- To store the object of same type in group
- Access using index
- Used in data structures
- Can be created on stack and heap & Access using dot operator

```
class Student{
    int rollNo;
    string name;

public:
    void acceptRecord(){
        cout << "Enter Roll No: " << endl;
        cin >> rollNo;
        cout << "Enter Name: " << endl;
        cin >> name;
    }

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

};

int main(){

    Student s_arr[3];  // array of object -> stack
//  Student s_arr1[] = new Student[3];  // array of object  -> heap
```

```
    for(int i=0;i<3;i++){
        s_arr[i].acceptRecord();
    }

    cout << "============Student Details============" << endl;

    for(int i=0;i<3;i++){
        s_arr[i].printRecord();
    }

    return 0;
}
```



Destructor

- It is called when your program goes out of scope
- It is used to release the memory allocated on heap
- It is called automatically, no need to call explicitly
- ~ used to define destructor
- No return_type, no parameters
- You cannot overload
- You can write code for releasing your used resource

```
class Student{
    int rollNo;
    string name;
```

```
public:

    Student() : rollNo(0), name(""){ //explicit default cons
        cout << "Constructor called!" << endl;
    }

    void acceptRecord(){
        cout << "Enter Roll No: " << endl;
        cin >> rollNo;
        cout << "Enter Name: " << endl;
        cin >> name;
    }

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

    ~ Student(){  //destructor
        cout << "Destructor called!" << endl;
    }
};

int main(){
//      Student s;
//  Student s_arr[3];  // array of object -> stack
    Student *s_arr1 = new Student[3];   // array of object  -> heap

//  delete[] s_arr1;        // to release the resources on heap -> array of object
//  delete = s;         // for single object

    return 0;
}
```

File handling in C++

- It is used to store the data permanently on disk
- fstream header file
- ifstream, ofstream, fstream classes
- ifstream -> Read operations
- ofstream -> Write operations
- fstream -> both
- open(), close() member functions
- getline() function to read line by line

```
#include <iostream>
#include <fstream> //header for file handling classes
using namespace std;

int main(){
```

```
    ofstream out;

    out.open("demo.txt");

    out << "Hi this is me!" << endl;
    out << "How are you today?" << endl;

    out.close(); //releasing the resource

    // for reading the file content
    ifstream in;
    string line;

    in.open("demo.txt");

    cout << "File content : " << endl;
    while(getline(in,line)){
        cout << line << endl;
    }
    in.close();

    return 0;
}
```

```
//typedef
typedef class StudentAttendanceReport{
public:
    int rollNo;
    string name;
    //mutable keyword
    mutable int attendanceCount = 0;        //to make this variable mutable, which
can change in const member function also

    const StudentAttendanceReport& joined_for_class() const {
        cout << "Roll No: " << rollNo << " "  << "Name: " << name << " " << "is
joined." << endl;
        return *this;
    }

    //const member function
    const StudentAttendanceReport& camera_on() const {
        attendanceCount++;
        cout << rollNo << " cam ON." << endl;
        return *this;
    }

    int getAttedanceCount() const{
        return attendanceCount;
    }

} SAR; //changed with typedef
```

```
//logger class for file logs
class Logger{
    ofstream out; // write file object created
public:

    //opened the file
    Logger(){
        out.open("attendance_log.txt");
    }

    // member function having bussiness logic
    void log(string msg){
        out << msg << endl;
    }

    // close the file
    ~ Logger(){
        out.close();
    }
};


int main() {
    SAR s1;
    Logger logger;

    s1.rollNo = 100;
    s1.name = "shil";

    logger.log("Attendance Report:");
    for(int i=1;i<=50;i++){
        if(i % 2 == 0){
            logger.log("Camera OFF!");
            s1.joined_for_class();
        }
        else{
            logger.log("Camera ON.");
            s1.joined_for_class().camera_on();
        }
    }
    logger.log("Attendance Count:" + to_string(s1.getAttedanceCount()));
    cout << "Attendace Count: " << s1.getAttedanceCount() << endl; // attendance
count
    return 0;
}
```

String class (User defined)

- To manage string data efficiently
- dynamic memory allocation using char*
- constructor to initialize the string

- destructor to release the memory
- member functions to accept and print the string

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

class String{
    char* str;
    int len;

    // private member function -> helper function
    void setString(const char* s){
        len = strlen(s);
        str = new char[len];
        strcpy(str, s);
    }

public:
    // default constructor
    String() : str(nullptr), len(0){
    }

    // parameterized constructor
    String(char* s, int len){
        setString(s);
    }

    // member function to accept string
    void accept(){
        char buffer[100];
        cout << "Enter string:" << endl;
        cin >> buffer;

        setString(buffer);
    }

    // member function to print string
    void print() const{
        cout << str << endl;
    }

    // destructor to release memory
    ~ String(){
        delete[] str;
    }
};

int main() {
    String s1;

    s1.accept();
```

```
    cout << "Your String:" << endl;
    s1.print();

    return 0;
}
```

Shallow copy vs Deep copy (using String class)

Shallow copy

- copies the address of the data members
- both objects point to the same memory location

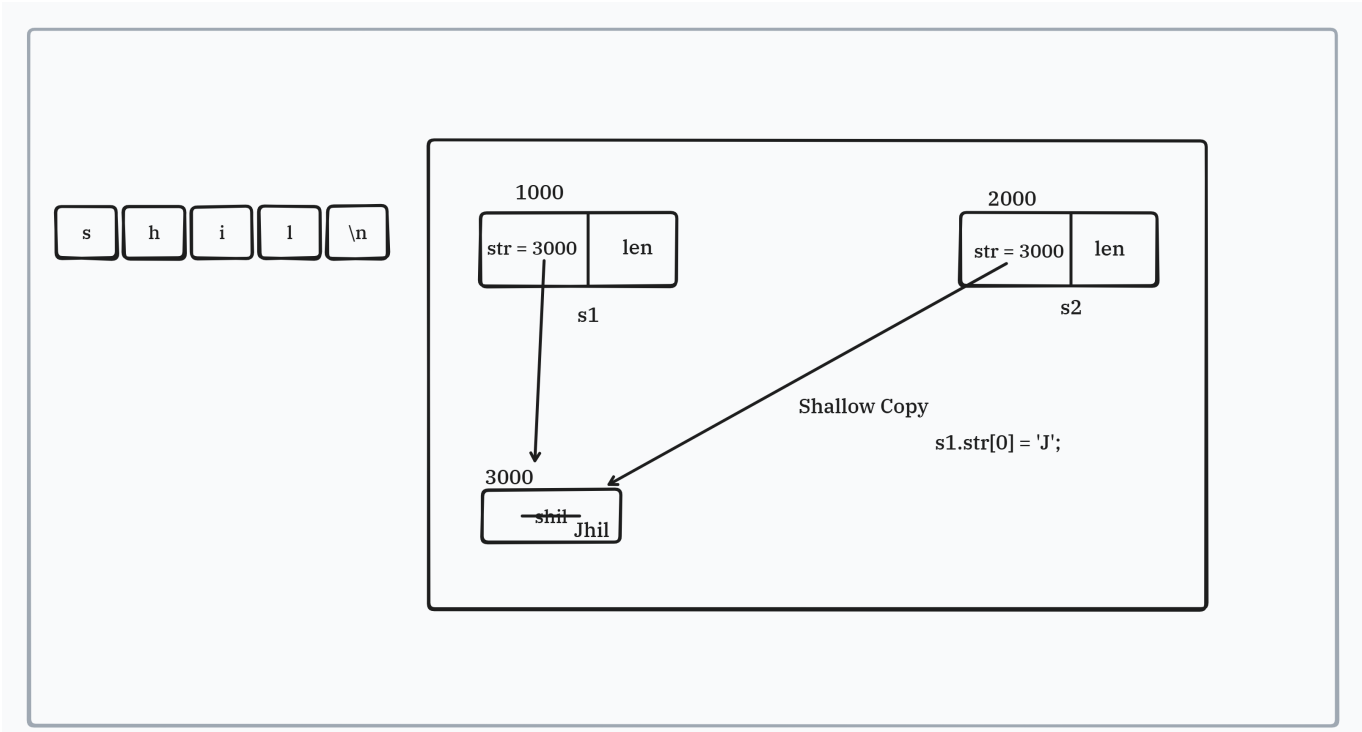
Deep copy

- creates a new copy of the data members
- copies the actual data to the new memory location

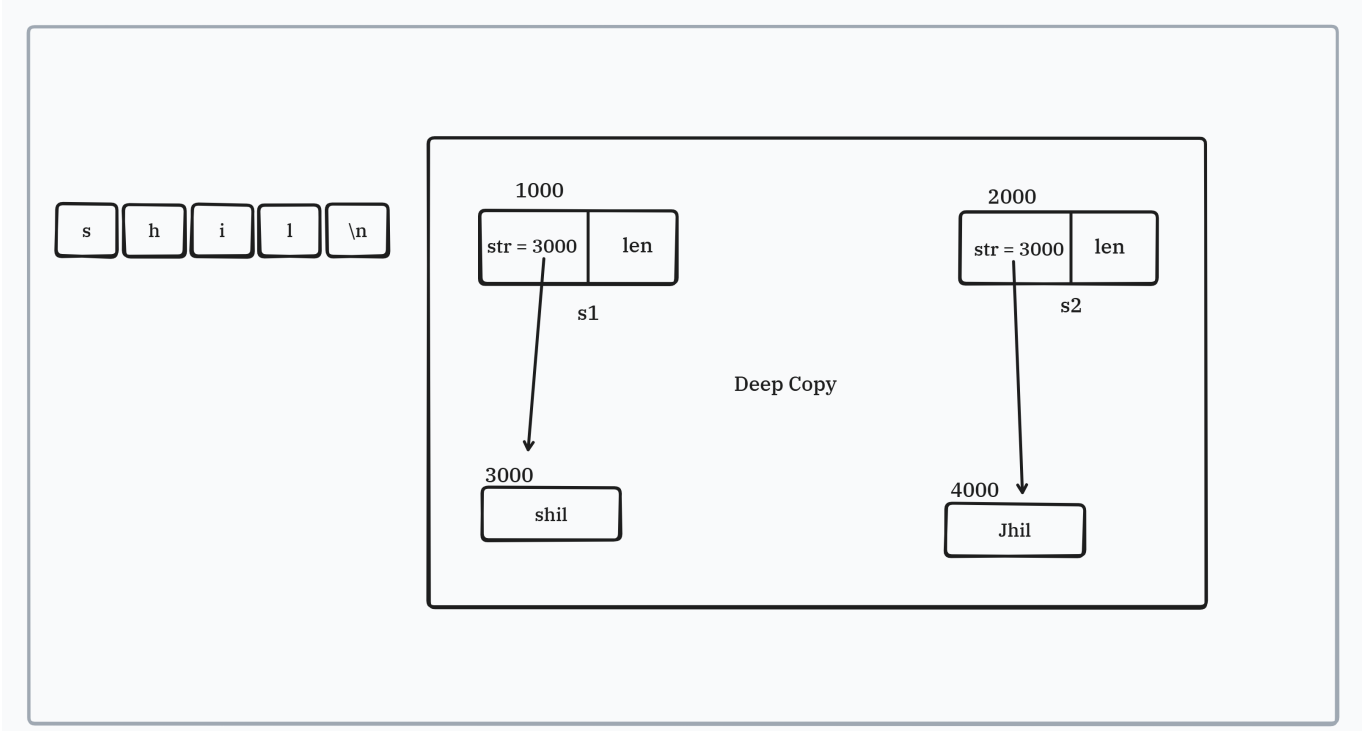
```
#include <iostream>
#include <cstring>
using namespace std;

class String{
public:
    char* str;
    int len;
    void setString(const char* s){
        len = strlen(s);
        str = new char[len];
        strcpy(str, s);
    }

    String() : str(nullptr), len(0){
    }

    String(char* s, int len){
        setString(s);
    }

//  default copy constructor -> shallow copy


//user defined copy constructor -> deep copy
//  String(String &other){
//      len = other.len;
//      str = new char[len];
//      strcpy(str, other.str);
//  }

    void accept(){
        char buffer[100];
```

```
        cout << "Enter string:" << endl;
        cin >> buffer;

        setString(buffer);
    }

    void print() const{
        cout << str << endl;
    }

    char* getStr() const {
        return str;
    }

    ~ String(){
        delete[] str;
    }
};

int main() {
    String s1;

    s1.accept();
    cout << "Your String:" << endl;
    s1.print();

//  String s2 = s1;     //using copy constructor -> compiler -> copy cons ->
shallow copy


    String s2 = s1;     //using own copy constructor -> deep copy

    cout <<"after assigment:" << endl;

    s2.print();

    cout << "S1 address: " << &s1 << endl;
    cout << "S2 address: " << &s2 << endl;

    s2.str[0] = 'J';

    cout <<"after changing s2 0th char element:" << endl;

    s1.print();
    s2.print();

    cout << (void*)s1.getStr() << endl;
    cout << (void*)s2.getStr() << endl;
    return 0;
}
```

Shallow Copy



Deep Copy



Copy constructor (and Rule of 3 intuition)

- It is used to initialize an object using another object of same class
- It is called when:
  - An object is passed by value to a function
  - An object is returned by value from a function
  - An object is explicitly copied

Rule of 3

- If a class requires a user-defined destructor, a user-defined copy constructor, or a user-defined copy assignment operator, it almost certainly requires all three.
- This is because these three functions are all responsible for managing the resources of the class, and if one of them is not implemented correctly, it can lead to memory leaks or other issues.

Tomorrow's Topics

- Matrix class