

Day 7 (08-12-2025)

Matrix class

- Dynamic memory allocation for 2D array
- Shallow copy vs Deep copy
- Working with double pointer
- Constructor to allocate memory
- Destructor to deallocate memory

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

class Matrix{
    int rows, cols;
    int **data;

public:

    Matrix(int r, int c) : rows(r), cols(c) {
        data = new int *[rows];

        for(int i=0;i<rows;i++){
            data[i] = new int[cols]; // coloumns
            for(int j=0;j<cols;j++){
                data[i][j] = 0;      //initialize with 0
            }
        }
    }

    //own copy constructor Matrix(Matrix &mat){} // deep copy
    ~ Matrix(){
        for(int i=0;i<rows;i++){
            delete[] data[i];
        }
        delete[] data;
    }

    // accept member functions
    void accept(){
        cout << "Enter Matrix : " <<endl;
        for(int i=0;i<rows;i++){
            for(int j=0;j<cols;j++){
                cin >> data[i][j];
            }
        }
    }

    // print member function
    void print(){
        cout << "====The Matrix====" << endl;
    }
}
```

```

        for(int i=0;i<rows;i++){
            for(int j=0;j<cols;j++){
                cout << data[i][j] << " ";
            }
            cout << endl;
        }
    }

};

int main() {
    Matrix m1(2,2);

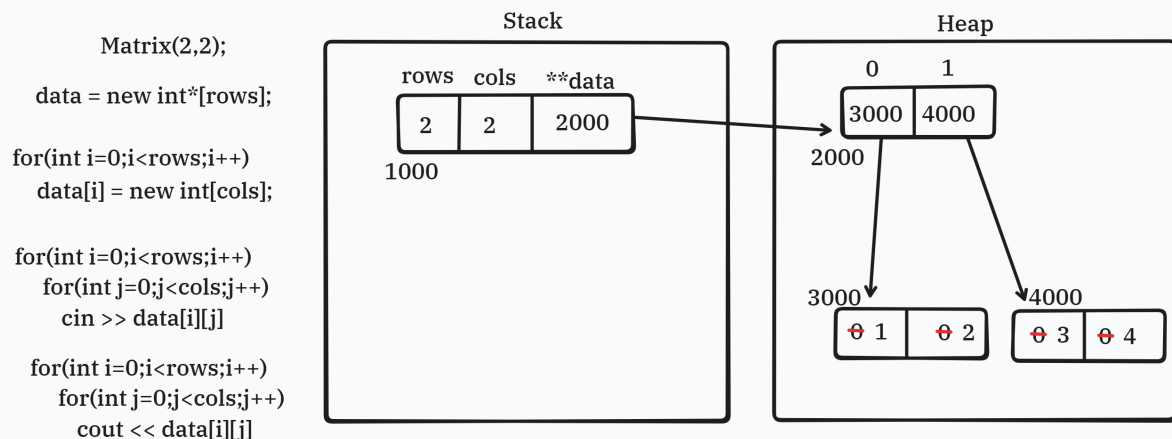
    Matrix m2 = m1;    //copy constructor -> shallow copy

    m1.accept();
    m1.print();

    m2.print();

    return 0;
}

```



Exception basics

- Exception is runtime problem which interrupts the flow of program
- In JAVA -> try, catch, finally, throw, throws
- In C++ -> try, catch, throw, throw(), noexcept
- try block -> this will have the risky code which can throw the exception

- catch block -> it will have the code to handle the exception
- throw -> to throw the exception to be caught in the catch block.
- throw() -> defined after function name () to denote it if it throws or does not throw any exception

try-catch syntax

```
try{
    // code which may throw exception
} catch(type1 e1){
    // code to handle exception of type1
} catch(type2 e2){
    // code to handle exception of type2
} catch(...){
    // code to handle any type of exception
}
```

throw syntax

```
throw exception_object; // can throw any type of object like int, float, string,
class object
```

catch variants

```
catch(type e)           // catch by value
catch(type &e)          // catch by reference
catch(const type &e)    // catch by const reference
catch(...)              // catch all type of exception
```

catch by reference is preferred to avoid slicing problem

Exception specification list

- Exception specification -> to specify whether function will throw exception or not
- noexcept -> this is modern syntax for throw() -> causing lot of problem -> c++11 they introduced noexcept
- noexcept : It denotes that this function will not throw any exception -> destructor -> program will terminate abnormally.
- noexcept(true) : It denotes that this function will not throw any exception -> same as noexcept
- noexcept(false) : It denotes that this function can throw exception -> same as no noexcept
- deprecated throw specifications
 - throw() : It denotes that this function will not throw any exception -> deprecated in c++11

- `throw(type)` : It denotes that this function will throw only specified type of exception -> deprecated in c++11
- `throw(...)` : It denotes that this function can throw any type of exception -> deprecated in c++11

throw() syntax

```
return_type function_name(parameters) throw(); // function will not throw any
exception
return_type function_name(parameters) throw(type1, type2); // function will throw
only specified type of exception
return_type function_name(parameters) throw(...); // function can throw any type
of exception
```

noexcept syntax

```
return_type function_name(parameters) noexcept; // function will not throw any
exception
return_type function_name(parameters) noexcept(true); // function will not throw
any exception
return_type function_name(parameters) noexcept(false); // function can throw
exception
```

- `noexcept` is preferred over `throw()` as it is more efficient and flexible - c++11 onwards

Example

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

//void divide(int a, int b) throw(){ // throws -> int value, throws() -> default
no exception
//void divide(int a, int b) noexcept { // denotes that no exception will occur
within this function

void divide(int a, int b) {
    if(b == 0){
        cerr << "Divide by Zero!" << endl;
        throw 10.5;    //string
    }
    int result = a / b;
    cout << result << endl;
}

int main() {

    int a = 10;
    int b = 0;    //divide by 0 -> terminating abnormally
```

```

try{
    divide(a,b);        // func call
} catch(int e){
    cerr << "Int Exception cause: " << e << endl;
} catch(char const* e){
    cerr << "Char* Exception cause: " << e << endl;
}
catch(...){
    cerr << "default exception catch" << endl;
}
cout << "continue the program" << endl;

return 0;
}

```

Nested try-catch

- try-catch block inside another try block
- inner try-catch will be executed first
- if exception is not handled in inner catch block, then it will be propagated to outer catch block

```

int main(){

    int a = 10;
    int b = 0;
    int res = 0;
    try{
        // throw "error: from outer try block";
        try{
            if(b == 0)
                throw "error: from inner try block";
            res = a / b;
        } catch(const char* e){
            cerr << "caught in inner catch block" << e << endl;
        }
        cout << "after try-catch block" << endl;
    } catch(...){
        cerr << "caught in outer catch block";
    }
    return 0;
}

```

Rethrowing

- throw; -> to rethrow the exception to be handled in outer catch block
- useful when inner catch block cannot handle the exception
- rethrowing will preserve the original exception type and value

```

void divide(int a, int b) {
    try{
        if(b == 0){
            cerr << "Divide by Zero!" << endl;
            throw 10.5;    //string
        }
        int result = a / b;
        cout << result << endl;
    } catch(double e){
        cout << "Error: in divide with code : " << e << endl;
        throw;    //rethrow -> to catch at main catch block
    }
}

int main() {
    int a = 10;
    int b = 0;    //divide by 0 -> terminating abnormally

    try{
        divide(a,b);    // func call
    } catch(int e){
        cerr << "Int Exception cause: " << e << endl;
    } catch(char const* e){
        cerr << "Char* Exception cause: " << e << endl;
    }
    catch(...){
        cerr << "default exception catch" << endl;
    }

    cout << "continue the program" << endl;

    return 0;
}

```

Custom exception class

- Create your own exception class by inheriting from `std::exception`
- Override `what()` method to provide custom error message
- Add additional information like line number, function name, file name
- Use macro **LINE, FUNCTION, FILE** to get current line number, function name, file name

Use of `source_location` library in c++20 for better stack trace (optional)

Example

```

#include <iostream>
#include <exception>    // for exception class
using namespace std;

```

```

class ArithmeticException : public exception{
    string message;
    int lineNumber;
    string functionName;
    string fileName;

public:
    ArithmeticException(string msg, int line, string func, string file) :
message(msg), lineNumber(line), functionName(func), fileName(file){

    }

    // with help of exception class
    // const char* what() const noexcept override{
    //     return message.c_str();
    // }

    void printStackTrace() const{
        cerr << message << endl;
        cerr << "at " << lineNumber << " of " << functionName << " " << fileName
<< endl;
    }

};

#define ARITH_EXCEPT(msg) ArithmeticException((msg), __LINE__, __FUNCTION__,
__FILE__) // define macro for exception

int divide(int a, int b){
    if(b == 0)
    //     throw ArithmeticException("Divide by zero!", __LINE__, __FUNCTION__,
__FILE__);
        throw ARITH_EXCEPT("Divide by Zero!"); // macro call(msg);
    return a / b;
}

int main() {
    int a = 10;
    int b = 0;

    try{
        cout << divide(a,b) << endl;
    }catch (ArithmeticException e) {
        e.printStackTrace();
    }

    return 0;
}

```

Friend function

- Function which can access private data members

- declared inside the class with friend keyword
- defined outside the class
- useful when we want to access private members of class without using getter/setter methods

Example

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

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 << rollNo << endl;
        cout << name << endl;
    }

    friend void getStudentDetails(Student &s);    //declare
};

void getStudentDetails(Student &s){            // definition
    cout << s.rollNo << " " << s.name << endl;    // valid
}

int main() {
    Student s1;

    s1.acceptRecord();
    s1.printRecord();

    // cout << s1.rollNo;    // not valid
    // cout << s1.name;    // not valid

    getStudentDetails(s1); // valid -> friend function

    return 0;
}
```

Friend class

- class which can access private data members of another class

- declared inside the class with friend keyword
- defined outside the class with friend keyword
- useful when two classes are closely related and need to access each other's private members

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

class Node{
    int data;
    Node* next;

public:
    Node(int d) : data(d), next(nullptr){

    }

    friend class LinkedList;    // whole access
};

class LinkedList{
    Node* head;

public:

    LinkedList(Node* h) : head(h) {

    }

    // insert(){
    // code for insertion
    // }

    void display(){
        Node* temp = head;
        while(temp != nullptr){
            cout << temp->data << " ";
            temp = temp->next;
        }
        cout << endl;
    }

    ~ LinkedList(){
        // code for deletion
    }

};

int main() {

    Node head(10);
    LinkedList list(&head);
```

```

    list.display();

    return 0;
}

```

Static member function

- static data member -> shared member for the whole class -> not for object
- static member function -> it is with your class
- static member function -> access static data member, cannot access normal data member
- called using class name :: operator
- cannot use this pointer
- useful for utility functions which are not related to object
- mostly used for counting number of objects created
-

```

#include <iostream>
using namespace std;

class Student{
    int rollNo;
    string name;
    static int count;    // static data member -> declaration of static variable
public:

    Student(){
        count++;
    }

    //static member function
    static void getCount(){
//        cout << rollNo << " " << name << endl;    //not valid -> error
        cout << "Static Function -> Total Students : " << count << endl;
    }

    void display(){
        cout << rollNo << " " << name << endl;
        cout << "Non Static Function -> Total Students : " << count << endl;
    }
};

int Student::count = 0;    //definition of static variable

int main() {
    Student s1;
    Student s2;
    Student s3;

    // s1.getCount(); // valid -> but not recommended;

```

```
s1.display();  
Student::getCount(); // valid -> recommended;  
  
return 0;  
}
```