Objectives :

- Object Oriented Programming and Namespace
- Handling Errors and Raising Exception

Object Oriented Programming :

Introduction :

In the last lecture, we explored **Functional Programming** and how it helps us restructure our programs into reusable sub-functions, dividing the program into smaller, manageable tasks. In this lecture, we will delve into **Object-Oriented Programming (OOP)**, another programming paradigm that focuses on transforming parts of a problem into **objects**. These objects represent real-world entities, such as cars, animals, or bank accounts.
Objects contain both **data (attributes)** and **code (methods)** that operate on that data. To create these objects, we need a **blueprint** that defines their structure, and this is where a **class** comes into play.

- **Objects**: Represent real-world entities, such as a car, a person, or a bank account. Each object has unique characteristics (**attributes**) and behaviors (**methods**).
- **Classes**: Serve as blueprints or templates for creating objects. A class defines the attributes and methods that objects of that class will have.

Creating Our First Class:

We create classes in the functions and constants sections of our program. To define a class, we use the `class` keyword followed by the name of the class and a pair of curly braces `{}`. Inside these braces, we write the body of the class, which includes its **methods** and **attributes**.
Before declaring these methods and attributes, we need to specify their **visibility** using access specifiers like `public`, `private`, or `protected`.

- **Attributes** are declared similarly to normal variables.
- **Methods** are declared like regular functions.

Example :

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

```
class Cat{
    public:
    string name = "catos";
    void talk(){
        cout << "meow, meow, meow" << endl;
    }
};
int main(){

     return 0;
}
```

We create **objects** from a class in the same way we create normal variables. We start with the **class name**, followed by the **object name**. Once the object is created, we can access its **methods** and **attributes** using the **dot notation (.)**.

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

class Cat{
    public:
    string name = "catos";
    void talk(){
        cout << "meow, meow, meow" << endl;
    }
};
int main(){
    Cat mycat;
    cout << mycat.name << endl;
    mycat.talk();
    return 0;
}
```

Methods and Attributes Visibility :

Before declaring attributes and methods, we need to set their visibility. Visibility determines where these attributes and methods can be accessed or modified.

Public :

A public method or attribute is visible and can be accessed or called both inside and outside the class.

Private :

A private method or attribute is accessible only inside the class. We cannot read or modify private attributes or call private methods outside the class body.

Protected :

Protected is similar to private in that it restricts access outside the class. However, the key difference is that a protected method or attribute can be accessed in a child class that inherits from the parent class.

Constructor and Destructor :

Constructor :

A constructor is a special class method that runs automatically when we create an object. It is commonly used to initialize the object's attributes.

- The constructor always has the same name as the class.
- It does not have a return type, not even `void`.
- It can have parameters, which represent the arguments passed to the object during its creation. These parameters are often used to initialize the object's attributes.

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

class Cat{
    public:
    string Name ;
    Cat(string name){
        Name = name;
    }
    void talk(){
        cout << "meow, meow, meow" << endl;
    }
};
```

```
int main(){
    Cat mycat("Mimi");
    cout << mycat.Name << endl;
    mycat.talk();
    return 0;
}
```

Destructor :

Unlike a constructor, a **destructor** is a special method that runs automatically when an object is no longer needed or when it is deleted. Destructors are primarily used for **cleanup tasks**, such as releasing resources (e.g., memory, file handles, or network connections) that the object may have acquired during its lifetime.
Creating a destructor is similar to creating a constructor, with one key difference: when defining a destructor, we use the **class name prefixed with a tilde (`~`)**. Like constructors, destructors do not have a return type.

Example :

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

class Cat{
    public:
    string Name ;
    Cat(string name){
        Name = name;
    }
    ~Cat(){
        cout << "The object " << Name << " get deleted" << endl;
    }
    void talk(){
        cout << "meow, meow, meow" << endl;
    }
};
int main(){
    Cat* mycat = new Cat("Mimi");
    cout << mycat->Name << endl;
    mycat->talk();
    delete mycat;
    cout << "After deleting the object" << endl ;
    return 0;
}
```

OOP Principles :

Encapsulation :

Encapsulation, a fundamental principle of object-oriented programming (OOP), involves bundling data and the methods that operate on that data within a single unit, or object.

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

#define PI 3.14
class Circle{
    private :
    float Radius;

    public:
    Circle(float r){
        Radius = r;
    }
    float getArea(){
        return Radius * Radius * PI;
    }
};
int main(){
    Circle circle(10);
    cout << "Area is : " << circle.getArea() << endl;
    return 0;
}
```

Here, we used **encapsulation** to bundle the **radius** (attribute) and the **getArea** method (behavior) inside the object. This ensures that the data (radius) and the operations (calculating the area) are tightly coupled within the same unit

Abstraction :

Abstraction refrer refers to the concept of hiding complex implementation details and exposing only the essential features or functionalities of an object. Abstraction allows programmers to focus on what an object does rather than how it does it.

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

class Robot{
    private :
    float fuel;
    public:
    Robot(float f){
        fuel = f;
    }
    void mouve(){
        if (fuel>0){
            fuel--;
            cout << "Robot mouved" << endl;
        }else{
            cout << "Robot out of fuels" << endl;
        }


    }
    void charge(float f){
        fuel += f;
        cout << "charged with " << f << endl;
    }
};
int main(){
    Robot robot1(5);
    for (int i = 0; i<6; i++){
        robot1.mouve();
    }
    robot1.charge(2);
    return 0;
}
```

This example demonstrates **abstraction** by hiding the implementation details of the `move` and `charge` functions. Users interact with the `Robot` object through a **simplified interface**, without needing to know the internal workings of these functions

Inheritance :

Inheritance is a mechanism that allows a class (a child or derived class) to inherit the attributes and methods of another class (a parent or base class) without needing to redefine them. For example, consider creating a program to represent a pet store with both cats and dogs. Instead of creating separate, entirely independent classes for "Cat" and "Dog" (which would lead to code duplication since they share common characteristics), we can use inheritance.
We can define a parent class, "Pet," containing attributes and methods common to all pets (e.g., `name` , `age` , `eat()` , `sleep()`). Then, we can create "Cat" and "Dog" classes that *inherit* from the "Pet" class. This means the "Cat" and "Dog" classes automatically gain access to the "Pet" class's attributes and methods. They can also define their own specific attributes and methods (e.g., `meow()` for "Cat", `bark()` for "Dog") that are unique to their type. This approach promotes code reuse, reduces redundancy, and makes the code more organized and maintainable.

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

class Pet{
    protected :
    string name;
    int age;
    public:
    Pet(string Name,int Age){
        name = Name;
        age = Age;
    }
    void sleep(){
        cout << name <<" sleeping" <<endl;

    }
    void walk(){
        cout << name <<" walking" << endl;
    }
};
class Cat:public Pet{
    public:
    Cat(string Name, int Age) : Pet(Name, Age){}
    void talk(){
        cout << "meow meow meow" << endl;
    }
};
class Dog:public Pet{
    public:
    Dog(string Name, int Age) : Pet(Name, Age){}
    void talk(){
        cout << "woof woof woof" << endl;
    }
};
int main(){
    Dog doggo("Jack",8);
    Cat cat("Mimi",4);
    cat.talk();
    cat.sleep();
    doggo.walk();
    return 0;
}
```

The `Dog` and `Cat` classes inherit the **attributes** (`name` and `age`) and **methods** (`sleep` and `walk`) from the `Pet` class.

Remarque :

You don't directly "inherit" a constructor in the same way you inherit methods or data members. Instead, you *call* the parent class's constructor from the child class's constructor.

Polymorphism :

Polymorphism is the final principle of Object-Oriented Programming (OOP). It allows objects of different classes to be treated as objects of a **common type**, even though they may exhibit different behaviors for the same method call. This enables flexibility and extensibility in code, as the same interface can be used to represent different underlying forms (or implementations).

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

```
class Pet{
    protected :
    string name;
    int age;
    public:
    Pet(string Name,int Age){
        name = Name;
        age = Age;
    }
    void sleep(){
        cout << name <<" sleeping" <<endl;

    }
    void walk(){
        cout << name <<" walking" << endl;
    }
};
class Cat:public Pet{
    public:
    Cat(string Name, int Age) : Pet(Name, Age){}
    void talk(){
        cout << "meow meow meow" << endl;
    }
};
class Dog:public Pet{
    public:
    Dog(string Name, int Age) : Pet(Name, Age){}
    void talk(){
        cout << "woof woof woof" << endl;
    }
};
int main(){
    Dog doggo("Jack",8);
    Cat cat("Mimi",4);
    cat.talk();
    doggo.talk();
    return 0;
}
```

Both the `Cat` and `Dog` classes have the `talk` method, but the **output differs** based on the specific implementation in each class. This is a simple example of **polymorphism**, where the same method call (`talk`) produces different results depending on the object's type.

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

class Distance{
    public:
    float d;
    Distance(float x){
        d = x;

    }
    Distance(float x,float y){
        d = sqrt(x*x+y*y);
    }
    Distance(float x,float y,float z){
        d = sqrt(x*x+y*y+z*z);
    }
};

int main(){
    Distance d1(1);
    Distance d2(3,4);
    Distance d3(3,4,6);
    cout << "d1 :" << d1.d << endl;
    cout << "d2 :" << d2.d << endl;
    cout << "d3 :" << d3.d << endl;
    return 0;
}
```

This is another example of **polymorphism**. The constructor behavior changes depending on the arguments passed to it. In this case, there are **three versions of the constructor**, each handling different input scenarios. The same principle can be applied to **class methods**, where the behavior of a method can vary based on the arguments it receives.

Setter and Getter :

Setters and getters are methods used to control access to the private attributes of a class. They are a key part of encapsulation, one of the fundamental principles of Object-Oriented Programming (OOP).

- **Setter**: A method used to **set** or modify the value of a private attribute.
- **Getter**: A method used to **retrieve** the value of a private attribute.

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

class Cat{
    private:
    string Name ;
    public:
    Cat(string name){
        Name = name;
    }
    void talk(){
        cout << "meow, meow, meow" << endl;
    }
    void setName(string name){
        Name = name;
    }
    string getName(){
        return Name;
    }
};
int main(){
    Cat mycat("Mimi");

    cout << mycat.getName() << endl;
    mycat.talk();
    mycat.setName("kitty");
    cout << mycat.getName() << endl;
    return 0;
}
```

Operator Overloading :

In Object-Oriented Programming (OOP), we can modify how basic operators (such as comparison and arithmetic operators) behave for our objects. This is achieved by overloading them.
In C++, the most commonly overloaded operators include:
Arithmetic Operators:

- **+: operator+**()
- **-: operator-**()
- *: operator**()
- **/: operator/**()
- **%: operator%**()

  Comparison Operators:
- **==: operator==**()
- **!=: operator!=**()
- **<: operator<**()
- **>: operator>**()
- **<=: operator<=**()
- **>=: operator>=**()

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

class Point{
    public:
    float x,y;
    Point(float X,float Y){
        x=X;
        y=Y;
    }
    Point operator+(const Point& other){
       Point r(0,0);
       r.x=x+other.x;
       r.y=y+other.y;
       return r;
    }

};
int main(){
    Point p1(4,8);
    Point p2(2,2);
    Point p3 = p1 + p2;
    cout << "p3 (" << p3.x << ", " << p3.y << ")" << endl;
    return 0;
}
```

Namespace :

A namespace organizes and groups related code elements, such as variables, functions, and classes, within a named block. It helps prevent naming conflicts and enhances code maintainability. We can think of it as a container that holds a set of identifiers, ensuring they remain distinct from those in other namespaces.
To create a namespace, we use the `namespace` keyword followed by the name of the namespace and then `{}`. Inside the curly braces `{}`, we declare variables, constants, functions, or other elements that we want to group within that namespace.

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

namespace Math{
    const float PI = 3.14;
    float circle_area(float r){
        return r * r * PI;
    }
}
int main(){

    cout << "the area of circle that have radius of 5 m is : " <<
Math::circle_area(5) << " m" << endl;
    return 0;
}
```

Handling Errors :

Introduction

While writing programs, we may encounter errors that can cause the program to break or behave unexpectedly. Sometimes, we might overlook certain scenarios, such as invalid user input, which can lead to crashes or unintended behavior. To address these issues, C++ provides a way to catch and handle different types of errors effectively.

Type of Errors :

Compilation Errors: These errors occur in programming languages that require compilation. They prevent the code from being compiled successfully, often due to issues like missing semicolons, incorrect syntax, or undefined variables.
Syntax Errors: These errors happen when there is a mistake in the code's syntax. For example, misspelling a function or variable name, forgetting parentheses `()`, or using incorrect punctuation.
Runtime Errors: These errors occur while the program is running and can cause it to crash. They are often caused by unexpected user actions, such as entering invalid input or accessing out-of-bounds memory.
Logic Errors: These errors result from mistakes in the program's logic or conditions. They don't crash the program but cause it to behave differently than intended. For example, a loop might run too many times or a condition might not work as expected.
Arithmetic Errors: These errors are related to mathematical operations. They can occur due to incorrectly formatted equations, division by zero, or incorrect operator precedence (e.g., `+`, `-`, `*`, `/`).
Resource Errors: These errors happen when the program exceeds the available resources of the system, such as running out of memory, disk space, or exceeding CPU usage limits.

Catching and Handling Errors :

Some errors can be fixed before compilation, such as syntax or type-related issues. However, other errors depend on user input or runtime conditions, which cannot always be prevented in advance. To handle such issues, C++ provides the `try` and `catch` mechanism, allowing us to manage errors gracefully and ensure the program doesn't crash unexpectedly.

```
try{
    // code tat may cuz error
}catch(...){
    handling the error
}
```

Here, we place the code that might cause an error inside the `try` block. If an error occurs, it can be caught and handled using the `catch` block. Inside the parentheses of the `catch` block, we specify the type of error we want to catch. You can use multiple `catch` blocks to handle different types of errors. Additionally, `catch (...)` can be used to catch any type of error.

Raising Errors :

In C++, we can do more than just `try` and `catch`. We can also raise custom errors. For example, if we set the minimum age to 18 and the user enters a younger age, the program can raise an error. To do this, we use the `throw` keyword, followed by the error class type and the message we want to display inside it.
To use standard error classes, we need to include the `<stdexcept>` header.

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

int main() {
    int age;
    cout << "Enter your age : ";
    cin >> age;
    try {
        if (age < 18){
            throw out_of_range("Your are younger to be in");
        }else{
        cout << "Welcome !" << endl;
        }

    }
    catch (const invalid_argument& e) {
        cout << "Invalid argument: " << e.what() << endl;
    }
    catch (const out_of_range& e) {
        cout << "Out of range: " << e.what() << endl;
    }
    catch (const exception& e) {
        cout << "Generic error: " << e.what() << endl;
    }
    return 0;
}
```

Here, if we enter an age smaller than 18, the program will throw an `out_of_range` exception with the message, "You are too young to proceed."
We can also see that there are multiple `catch` blocks, each designed to handle a different type of exception.
The `.what()` method, which is part of the exception class, is used to display the exception message.

Creating Exceptions :

Exceptions are classes, so by using the OOP inheritance concept, we can create our own custom exceptions by inheriting from the base `exception` class. To create a custom exception, the first thing we need to do is include the `<exception>` header. Then, we create a class that inherits from the `exception` class.
The body of the cosumize exception classshould have the following :

- constructor that initialise the exception message
- ovveride the what function that display the excpetion message

Example :

```
#include <iostream>
#include <string>
#include <exception>
using namespace std;
class zerodivision : exception {
private:
    string message;
public:
    zerodivision(string msg)  {
    message = msg;
    }
    const char* what() const noexcept override {
        return message.c_str();
    }
};
int main() {
    float a, b;
    cout << "Enter a : ";
    cin >> a;
    cout << "Enter b: ";
    cin >> b;
    try {
        if (b == 0){
            throw zerodivision("you can't divide by 0");
        }else{
        cout << "a / b = " << a / b << endl;
        }
    }catch (const zerodivision& e) {
        cout << "\nError: " << e.what() << endl;
    }
    return 0;
}
```

The `const char* what() const noexcept override` method is a key part of the `exception` class in C++. It is used to provide a description of the error that occurred.

- **`const char*`** :
  - This is the return type of the `what()` method.
  - The string describes the error that occurred.
- **`what()`** :
  - This is the name of the method.
  - It is a virtual method in the `exception` class, which means it can be overridden by derived classes.
- **`const`** :
  - The `const` keyword at the end of the method declaration indicates that this method does not modify the state of the object.
  - It is a **const member function**, meaning it can be called on `const` objects.
- **`noexcept`** :
  - The `noexcept` keyword specifies that this method does not throw any exceptions.
  - It is a guarantee to the compiler and the programmer that this method will not throw exceptions, making it safe to use in exception-handling code.
- **`override`** :
  - The `override` keyword indicates that this method is overriding a virtual method from the base class (`exception`).
  - It ensures that the method signature matches exactly with the base class method. If it doesn't, the compiler will generate an error.

Tasks :

Task 1 :

Create a class for a bank management system. The class should represent a bank account and include the basic methods and attributes.

- **Attributes:**
  - `account_number` : A unique identifier for the account.
  - `account_holder_name` : The name of the account holder.

    - `account_balance`: The current balance in the account.
- **Methods:**
    - **Constructor**: Initializes the object attributes (account number, account holder name, and initial balance).
    - **Deposit**: Adds a specified amount of money to the account balance.
    - **Withdraw**: Deducts a specified amount of money from the account balance (if sufficient funds are available).
    - **Get Balance**: Returns the current balance of the account.
    - **Transfer**: Transfers a specified amount of money from one account to another.
    - **Display Account Info**: Displays the account information (account number, account holder name, and balance).

Task 2 :

Create a class for representing complex numbers. The class should include the basic attributes and methods, and it should support operations by overriding operators. Additionally, it should be able to perform comparisons and return the modulus of the complex number.

- **Attributes:**
    - `real`: The real part of the complex number.
    - `imaginary`: The imaginary part of the complex number.
- **Methods:**
    - **Constructor**: Initializes the object attributes (real and imaginary parts).
    - **Addition (`+`)**: Overrides the `+` operator to add two complex numbers.
    - **Subtraction (`-`)**: Overrides the `-` operator to subtract one complex number from another.
    - **Multiplication (`*`)**: Overrides the `*` operator to multiply two complex numbers.
    - **Division (`/`)**: Overrides the `/` operator to divide one complex number by another.
    - **Equality (`==`)**: Overrides the `==` operator to check if two complex numbers are equal.
    - **Modulus**: Calculates and returns the modulus (absolute value) of the complex number.
    - **Display**: Displays the complex number in the form `a + bi`.