Week 1: Classes and Objects

- **Classes**: Blueprints for creating objects. Define data (attributes) and methods (functions).
- **Objects**: Instances of a class, created using the class constructor.
- **Encapsulation**: Restricting access to certain components of objects.
- **Access Modifiers**:
    - public: Accessible from outside the class.
    - private: Only accessible within the class.
    - protected: Accessible within the class and derived classes.

```
class Person {
private:
   std::string name;
public:
   Person(std::string n) : name(n) {}
   void display() const { std::cout << name << std::endl; }
};
```

Week 2: Inheritance and Polymorphism

- **Inheritance**: Allows a new class (derived) to inherit attributes and methods from an existing class (base).
- **Base Class**: The class being inherited from.
- **Derived Class**: The class that inherits the properties of the base class.
- **Polymorphism**: Ability to treat objects of different classes in a uniform way.
- **Virtual Functions**: Functions in base classes that can be overridden in derived classes for runtime polymorphism.

```
class Animal {
public:
   virtual void speak() { std::cout << "Animal sound" << std::endl; }
};
class Dog : public Animal {
public:
   void speak() override { std::cout << "Woof" << std::endl; }
};
```

Week 3: Composition, Aggregation, and Association

- **Composition**: A strong "has-a" relationship where the contained objects' lifetime depends on the containing object.
- **Aggregation**: A weaker relationship; the contained object can exist independently of the container.
- **Association**: No ownership, just a relationship between two objects.

```
class Engine { // Example of Composition:
public:
   void start() { std::cout << "Engine started" << std::endl; }
};
class Car {
   Engine engine;  // Composition
public:
   void startCar() { engine.start(); }
};

class Club { Example of Aggregation:
   const Person* members[50]; // Aggregation: Person exists independently
   int memberCount = 0;
public:
   void addMember(const Person& p) { members[memberCount++] = &p; }
   void displayMembers() const {
      for (int i = 0; i < memberCount; ++i)
         std::cout << members[i]->getName() << std::endl;
   }
};
```

Week 4: Lambda Functions

- **Lambda Functions**: Anonymous inline functions, often used to define small operations.
- **Syntax**:
-

[capture-list](parameters) -> return-type { function-body } **EXAMPLE**:

```
auto add = [](int a, int b) -> int { return a + b; };
std::cout << add(5, 3) << std::endl; // Outputs 8
```

- **Capture Types**:
    - [=]: Capture by value (copies external variables into lambda).
    - [&]: Capture by reference (accesses external variables directly).

Example of Capturing by Reference:

```
int x = 10;
auto increment = [&]() { x++; };
increment();
std::cout << x << std::endl; // Outputs 11
```

Week 5: Exception Handling

- **try-catch block**: Mechanism to handle runtime errors (exceptions) thrown by functions.
- **throw**: Used to report an exception from a function.
- **noexcept**: Indicates that a function does not throw exceptions.

```
void divide(int a, int b) {
    if (b == 0)
        throw std::invalid_argument("Division by zero!");
    std::cout << a / b << std::endl;
}

int main() {
    try {
        divide(10, 0);
    } catch (const std::exception& e) {
        std::cout << e.what() << std::endl; // Outputs: Division by zero!
    }
}
```

- **Standard Exceptions**:
  - std::invalid_argument: Invalid argument provided.
  - std::out_of_range: Accessing elements outside bounds.
  - std::runtime_error: General runtime error.
- **Custom Exceptions**: You can also define your own exception classes by inheriting from std::exception.

```
class MyException : public std::exception {
public:
    const char* what() const noexcept override {
        return "My custom exception occurred!";
    }
};
```

Additional Concepts:

- **Rule of Three/Five**:
  - **Destructor**: Cleans up resources.
  - **Copy Constructor**: Creates a copy of an object.
  - **Copy Assignment Operator**: Assigns content of one object to another.
  - **Move Constructor**: Moves resources from one object to another.
  - **Move Assignment Operator**: Moves assignment of one object to another.

Example of Rule of Five:

```
class MyClass {
    int* data;
public:
    MyClass(size_t size) : data(new int[size]) {}
    ~MyClass() { delete[] data; }
    MyClass(const MyClass& other) { /*...*/ }
    MyClass& operator=(const MyClass& other) { /*...*/ return *this; }
    MyClass(MyClass&& other) noexcept : data(other.data) { other.data = nullptr; }
    MyClass& operator=(MyClass&& other) noexcept { /*...*/ return *this; }
};
```

Templates:

- **Templates**: Allow generic programming by enabling functions and classes to operate on different types.

Example:

```
template <typename T>
T maximum(T a, T b) {
    return (a > b) ? a : b;
}

int main() {
    std::cout << maximum(10, 20) << std::endl;  // Outputs 20
    std::cout << maximum(4.5, 2.3) << std::endl;  // Outputs 4.5
}
```
