|  |  |  |
| --- | --- | --- |
| **400** Bad Request | Invalid input | Validate request parameters |

|  |  |  |
| --- | --- | --- |
| **401** Unauthorized | Missing/invalid auth | Check JWT, API key, or session |

|  |  |  |
| --- | --- | --- |
| **403** Forbidden | No permission | Ensure correct roles/permissions |

|  |  |  |
| --- | --- | --- |
| **404** Not Found | Wrong URL/endpoint | Verify API route |

|  |  |  |
| --- | --- | --- |
| **405** Method Not Allowed | Wrong HTTP method | Use correct GET/POST/PUT/DELETE |

|  |  |  |
| --- | --- | --- |
| **408** Request Timeout | Slow response | Optimize queries, use caching |

|  |  |  |
| --- | --- | --- |
| **429** Too Many Requests | Rate limit exceeded | Implement retry/backoff strategy |

|  |  |  |
| --- | --- | --- |
| **500** Internal Server Error | Server crash | Debug logs, check exception handling |

|  |  |  |
| --- | --- | --- |
| **502** Bad Gateway | Proxy issue | Check server connectivity |

|  |  |  |
| --- | --- | --- |
| **503** Service Unavailable | Overloaded server | Scale infrastructure |

2. What are the types of testing performed on embedded systems?

Answer:

Unit Testing: Tests individual components or modules.

Integration Testing: Verifies interfaces between modules or hardware/software.

System Testing: Validates the system as a whole.

Regression Testing: Ensures new changes do not introduce defects.

Performance Testing: Checks system response time, resource usage, and efficiency.

Stress Testing: Evaluates the system's behavior under extreme conditions.

Real-Time Testing: Ensures the system meets timing constraints.

Module 1: Debugging Logical Errors in Loops

Bug:

The loop starts from i = 0, so factorial \*= i multiplies the result by 0, making the factorial incorrect.

Fix:

Start the loop with i = 1 since 0! = 1 and avoid multiplying by 0.

#include <stdio.h>

int main() {

int n = 5; // Calculate factorial of 5

int factorial = 1;

// Correct initialization of loop variable

for (int i = 1; i <= n; i++) {

factorial \*= i; // Multiply factorial by current number

}

printf("Factorial of %d is %d\n", n, factorial);

return 0;

}

Debugging Steps:

Use GDB:

gdb ./logical\_error

Set breakpoints and print variable values:

break main

run

print i

print factorial

Module 2: Debugging Null Pointer Dereference

Bug:

The condition if (n < 0) skips memory allocation, and the program attempts to dereference a NULL pointer (array[0]).

Fix:

Allocate memory if n > 0 and check for NULL pointers before dereferencing.

#include <stdio.h>

#include <stdlib.h>

int main() {

int \*array = NULL;

int n = 5;

// Allocate memory only for valid sizes

if (n > 0) {

array = malloc(n \* sizeof(int));

if (array == NULL) {

printf("Memory allocation failed.\n");

return -1;

}

} else {

printf("Invalid size.\n");

return -1;

}

// Safe dereference

array[0] = 10;

printf("Value at index 0: %d\n", array[0]);

free(array); // Free allocated memory

return 0;

}

Debugging Steps:

Use GDB:

gdb ./segmentation\_fault\_rd

break main

run

print array

Inspect memory allocation:

print n

print \*array

Module 3: Memory Corruption in Dynamic Arrays

Bug:

strcpy copies more data into a fixed-size buffer, leading to a buffer overflow.

Fix:

Use strncpy to copy data with a size limit, ensuring the buffer is null-terminated.

#include <stdio.h>

#include <string.h>

void safeFunction(char \*str) {

char buffer[10];

// Prevent buffer overflow with bounds checking

strncpy(buffer, str, sizeof(buffer) - 1);

buffer[sizeof(buffer) - 1] = '\0'; // Ensure null termination

printf("Buffer content: %s\n", buffer);

}

int main() {

char largeInput[] = "TooLongInputDataExceedingBuffer";

safeFunction(largeInput);

return 0;

}

Debugging Steps:

Use GDB:

gdb ./track\_memory\_corruption

break safeFunction

run

Observe memory:

x/20x &buffer

Module 4: Debugging Race Conditions in Multithreaded Code

Bug:

Threads access the shared\_counter simultaneously without synchronization, leading to race conditions.

Fix:

Use pthread\_mutex\_lock and pthread\_mutex\_unlock to ensure synchronized access.

#include <stdio.h>

#include <pthread.h>

int shared\_counter = 0; // Shared variable

pthread\_mutex\_t lock; // Mutex lock

void \*increment(void \*arg) {

for (int i = 0; i < 10; i++) {

pthread\_mutex\_lock(&lock);

shared\_counter++; // Safely increment

printf("Incrementing: Counter = %d\n", shared\_counter);

pthread\_mutex\_unlock(&lock);

}

return NULL;

}

void \*decrement(void \*arg) {

for (int i = 0; i < 10; i++) {

pthread\_mutex\_lock(&lock);

shared\_counter--; // Safely decrement

printf("Decrementing: Counter = %d\n", shared\_counter);

pthread\_mutex\_unlock(&lock);

}

return NULL;

}

int main() {

pthread\_t thread1, thread2;

pthread\_mutex\_init(&lock, NULL); // Initialize mutex

pthread\_create(&thread1, NULL, increment, NULL);

pthread\_create(&thread2, NULL, decrement, NULL);

pthread\_join(thread1, NULL);

pthread\_join(thread2, NULL);

pthread\_mutex\_destroy(&lock); // Destroy mutex

printf("Final Counter Value: %d\n", shared\_counter);

return 0;

}

Debugging Steps:

Compile with thread support:

Use GDB:

gdb ./bug\_raceconditions\_multithread

info threads

Module 5: Recursive Function Debugging

Bug:

Missing base case causes infinite recursion and stack overflow.

Fix:

Add a base case to terminate the recursion.

#include <stdio.h>

void fixedRecursion(int n) {

if (n > 10) return; // Base case

printf("Recursion level: %d\n", n);

fixedRecursion(n + 1); // Recursive call

}

int main() {

fixedRecursion(1);

return 0;

}

Debugging Steps:

Use GDB:

gdb ./error\_recursive\_logic

break fixedRecursion

run

backtrace # Inspect call stack

|  |  |
| --- | --- |
| **Locators** | Find elements using ID, Name, XPath, etc. |

|  |  |
| --- | --- |
| **Interactions** | Click, Type, Clear, Submit |

|  |  |
| --- | --- |
| **Waits** | Implicit & Explicit Waits |

|  |  |
| --- | --- |
| **Alerts & Windows** | Handle popups, frames, and tabs |

|  |  |
| --- | --- |
| **Mouse Actions** | Hover, Drag & Drop |

|  |  |
| --- | --- |
| **Screenshots** | Capture webpage screenshots |

|  |  |
| --- | --- |
| **PyTest** | Write, Run, and Report test cases |

|  |  |
| --- | --- |
| **Headless Mode** | Run tests without GUI |

|  |  |
| --- | --- |
| **Parallel Execution** | Run tests faster with -n option |

|  |  |
| --- | --- |
| **CI/CD Integration** | Automate tests using Jenkins |

 **REST** stands for **Representational State Transfer**.

 REST API is a web service architecture that uses standard HTTP methods to perform CRUD operations.

 **Resources** are represented as URLs (Uniform Resource Identifiers - URIs).

| **Method** | **Action** | **Example** |
| --- | --- | --- |

|  |  |  |
| --- | --- | --- |
| **GET** | Retrieve data (Read) | GET /api/items |

|  |  |  |
| --- | --- | --- |
| **POST** | Create new data | POST /api/items |

|  |  |  |
| --- | --- | --- |
| **PUT** | Update/Replace existing data | PUT /api/items/1 |

|  |  |  |
| --- | --- | --- |
| **PATCH** | Update/Modify partial data | PATCH /api/items/1 |

|  |  |  |
| --- | --- | --- |
| **DELETE** | Remove data | DELETE /api/items/1 |

| **Code** | **Meaning** | **Description** |
| --- | --- | --- |
| **200** | OK | Request succeeded (GET, PUT, PATCH). |
| **201** | Created | Resource created successfully (POST). |
| **204** | No Content | Request succeeded, no content to return. |
| **400** | Bad Request | Invalid input from client. |
| **401** | Unauthorized | Authentication required. |
| **403** | Forbidden | Access to the resource is denied. |
| **404** | Not Found | Resource does not exist. |
| **500** | Internal Server Error | Server encountered an unexpected condition. |

**Authentication in REST API**

* **API Key**: A unique key for accessing the API.
* **OAuth**: Token-based authentication.
* **JWT (JSON Web Tokens)**: Secure and stateless token for API authentication.

Good Morning Ma’am,

First of all thanks for giving me this wonderfull opportunity to introduce me in front of you.

My name is **Manish Kumar Singh**, am from Gopalganj, Bihar.

And my highest qualification is in 2022 I have be completed my B.Tech with specialization in CSE, from GNDEC Ludhiana, Punjab. reason behind choosing the CSE because of the interest and i want build my career in Software Development.

Now apart from this i completed some technical skills courses and get certification which is Python, Django from GeeksforGeeks, SQL from Hackerrank, UML & OOD and Active Listening from Udemy.

Currently am undergoing with training at RPS Consulting for Wipro organization. During this training. I learned the fundamentals of Python, Linux, Testing Fundamentals, and C Programming. Additionally now Soft skills training is going on to enhance our personalities, decision making, and professional communications.

And I love working with computers and solving logical problem.

My hobbies are playing outdoor games.

Thank You…

 **Class**: A blueprint for creating objects, encapsulating data and behavior.

 **Object**: An instance of a class representing real-world entities with properties and methods.

 **Abstract Class**: A class that cannot be instantiated and defines methods to be implemented by subclasses.

 **Interface**: A contract specifying methods that must be implemented by a class.

 **Encapsulation**: Hiding internal details of an object and exposing only necessary functionalities.

 **Polymorphism**: The ability of objects or methods to take multiple forms.

 **Constructor**: A special method used to initialize objects when a class is instantiated.

 **Abstract Method**: A method declared in an abstract class but without an implementation.

 **Method Overloading**: Defining multiple methods in a class with the same name but different parameters.

 **Method Overriding**: Redefining a method in a subclass that exists in the parent class.

 **Inheritance**: The mechanism where one class derives properties and behavior from another class.

 **Regular Expression**: A pattern used to match and manipulate text efficiently.

 **File Handling**: Managing files (create, read, write, delete) in a program.

 **Exception Handling**: Managing errors in a program gracefully using try-except blocks.

| **Feature** | **Generator** | **Decorator** |
| --- | --- | --- |
| **Purpose** | Generates a sequence of values lazily using yield. | Modifies or extends the behavior of functions/methods. |
| **Return Type** | Returns an iterator. | Returns a wrapped function. |
| **Keyword Used** | Uses the yield keyword. | Uses the @decorator syntax (optional). |

 **Pytest**: A Python testing framework for writing and running test cases.

 **Fixture**: A Pytest feature for providing setup and teardown code for tests.

 **Markup**: Text annotated with tags to define structure or presentation (e.g., HTML).

 **BDD Framework**: A tool to support Behavior-Driven Development for testing based on user stories.

|  |  |
| --- | --- |
| **Given** | Describes the initial context or precondition. |
| **When** | Describes the action or event. |
| **Then** | Describes the expected outcome. |
| **And**/**But** | Additional steps to extend scenarios; |

* It is a software development approach that encourages collaboration between developers, testers, and non-technical stakeholders.
* Focuses on **defining application behavior** using plain, human-readable language.

 **REST API**: A web API using HTTP methods for interaction, adhering to REST principles.

|  |  |  |
| --- | --- | --- |
| **GET** | Retrieve data (Read) | GET /api/items |

|  |  |  |
| --- | --- | --- |
| **POST** | Create new data | POST /api/items |

|  |  |  |
| --- | --- | --- |
| **PUT** | Update/Replace existing data | PUT /api/items/1 |

|  |  |  |
| --- | --- | --- |
| **PATCH** | Update/Modify partial data | PATCH /api/items/1 |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **DELETE** | | Remove data | DELETE /api/items/1 | |
| **200** | | OK | | | Request succeeded (GET, PUT, PATCH). | |
| **201** | | Created | | | Resource created successfully (POST). | |
| **204** | | No Content | | | Request succeeded, no content to return. | |
| **400** | | Bad Request | | | Invalid input from client. | |
| **401** | | Unauthorized | | | Authentication required. | |
| **403** | | Forbidden | | | Access to the resource is denied. | |
| **404** | | Not Found | | | Resource does not exist. | |
| **500** | | Internal Server Error | | | Server encountered an unexpected condition. | |
|  | |  |  | |

 **SOAP**: A protocol for exchanging structured information in web services using XML.

 **Status Code**: HTTP codes indicating the result of a request (e.g., 200 OK, 404 Not Found).

 **OAuth**: A framework for delegated access to resources without sharing credentials.

 **JWT**: A compact, secure token format for transmitting information between parties.

* **API Key**: A unique key for accessing the API.
* **OAuth**: Token-based authentication.
* **JWT (JSON Web Tokens)**: Secure and stateless token for API authentication.

| **Feature** | **REST** | **SOAP** |
| --- | --- | --- |

|  |  |  |
| --- | --- | --- |
| **Protocol** | HTTP | HTTP, SMTP, TCP |

|  |  |  |
| --- | --- | --- |
| **Data Format** | JSON, XML | XML |

|  |  |  |
| --- | --- | --- |
| **Flexibility** | More flexible | More rigid |

|  |  |  |
| --- | --- | --- |
| **Ease of Use** | Easier to implement | More complex |

| **Feature** | **List** | **Tuple** | **Dictionary** | **Set** |
| --- | --- | --- | --- | --- |
| **Definition** | Ordered, mutable collection of items | Ordered, immutable collection of items | Unordered, key-value pairs | Unordered, mutable collection of unique items |
| **Mutable/Immutable** | Mutable (can change) | Immutable (cannot change) | Mutable (keys can be added/removed) | Mutable (items can be added/removed) |
| **Duplicates Allowed** | Yes | Yes | Keys: No; Values: Yes | No |
| **Ordering** | Preserves order | Preserves order | Keys are unordered | Unordered |
| **Indexing** | Yes (supports indexing and slicing) | Yes (supports indexing and slicing) | No (keys act as indices) | No |
| **Usage** | General-purpose collection of items | Fixed collection of items | Mapping keys to values | Unique collection of items |
| **Syntax** | [item1, item2, ...] | (item1, item2, ...) | {key1: value1, key2: value2, ...} | {item1, item2, ...} |
| **Example** | [1, 2, 3, 4] | (1, 2, 3, 4) | {'a': 1, 'b': 2} | {1, 2, 3, 4} |

| **Feature** | **Abstract Class** | **Interface** |
| --- | --- | --- |

|  |  |  |
| --- | --- | --- |
| **Definition** | A class that can have both abstract methods and concrete methods (implemented methods). | A blueprint that defines only abstract methods with no concrete implementations. |

|  |  |  |
| --- | --- | --- |
| **Purpose** | Used when classes share some common functionality but also need specific behavior. | Used to define a strict contract that implementing classes must adhere to. |

|  |  |  |
| --- | --- | --- |
| **Implementation** | Defined using the abc module and the @abstractmethod decorator. | Also defined using the abc module and the @abstractmethod decorator. |

|  |  |  |
| --- | --- | --- |
| **Concrete Methods** | Can include concrete (implemented) methods. | Cannot include concrete methods. |

|  |  |  |
| --- | --- | --- |
| **Attributes** | Can have instance variables and static methods. | Usually focuses only on method declarations. |

|  |  |  |
| --- | --- | --- |
| **Instantiation** | Cannot be instantiated directly. | Cannot be instantiated directly. |

|  |  |  |
| --- | --- | --- |
| **Multiple Inheritance** | Supports single or multiple inheritance. | Primarily used in multiple inheritance for implementing multiple interfaces. |

|  |  |  |
| --- | --- | --- |
| **When to Use** | Use when you need to share common functionality across related classes. | Use when you need unrelated classes to adhere to a specific contract. |

**Abstract Example:-**

from abc import ABC, abstractmethod

class Shape(ABC):

@abstractmethod

def area(self):

pass

def describe(self):

return "I am a shape."

class Circle(Shape):

def \_\_init\_\_(self, radius):

self.radius = radius

def area(self):

return 3.14 \* self.radius \* self.radius

circle = Circle(5)

print(circle.area()) # Output: 78.5

print(circle.describe()) # Output: I am a shape.

**Interface……….>**

from abc import ABC, abstractmethod

class Animal(ABC):

@abstractmethod

def sound(self):

pass

class Dog(Animal):

def sound(self):

return "Bark"

class Cat(Animal):

def sound(self):

return "Meow"

dog = Dog()

cat = Cat()

print(dog.sound()) # Output: Bark

print(cat.sound()) # Output: Meow

| **Feature** | **Method Overloading** | **Method Overriding** |
| --- | --- | --- |
| **Definition** | Same method name, different parameters within the same class. | Same method name, same parameters, different implementation in subclass. |
| **Inheritance** | Not related to inheritance. | Always involves inheritance. |
| **Purpose** | Achieve method functionality based on input arguments. | Modify or extend the behavior of a parent class method. |
| **Example** | One method handling different argument combinations. | Subclass redefining a method of the parent class. |

**Method overloading:……..**

class Calculator:

def add(self, a, b=0, c=0):

return a + b + c

calc = Calculator()

print(calc.add(5)) # Output: 5 (only one argument)

print(calc.add(5, 10)) # Output: 15 (two arguments)

print(calc.add(5, 10, 15)) # Output: 30 (three arguments)

**Method overriding:……**

class Parent:

def greet(self):

return "Hello from Parent!"

class Child(Parent):

def greet(self):

return "Hello from Child!"

parent = Parent()

child = Child()

print(parent.greet()) # Output: Hello from Parent!

print(child.greet()) # Output: Hello from Child!

| **Feature** | **Generator** | **Decorator** |
| --- | --- | --- |
| **Purpose** | Generates a sequence of values lazily using yield. | Modifies or extends the behavior of functions/methods. |
| **Return Type** | Returns an iterator. | Returns a wrapped function. |
| **Keyword Used** | Uses the yield keyword. | Uses the @decorator syntax (optional). |

**Generator**

* **Definition**: A generator is a special type of function that returns an iterator and generates values on the fly using the yield keyword.
* **Use Case**: Useful for generating a sequence of values without storing the entire sequence in memory.

def fibonacci(n):

a, b = 0, 1

for \_ in range(n):

yield a

a, b = b, a + b

for num in fibonacci(5):

print(num) # Output: 0, 1, 1, 2, 3

**Decorator**

* **Definition**: A decorator is a function that takes another function as input and extends or modifies its behavior without modifying the original function itself.
* **Use Case**: Useful for adding functionality like logging, access control, or timing to functions.

def my\_decorator(func):

def wrapper():

print("Something before the function.")

func()

print("Something after the function.")

return wrapper

@my\_decorator

def say\_hello():

print("Hello!")

say\_hello()

# Output:

# Something before the function.

# Hello!

# Something after the function.

 **REST** stands for **Representational State Transfer**.

 REST API is a web service architecture that uses standard HTTP methods to perform CRUD operations.

 **Resources** are represented as URLs (Uniform Resource Identifiers - URIs).

| **Method** | **Action** | **Example** |
| --- | --- | --- |

|  |  |  |
| --- | --- | --- |
| **GET** | Retrieve data (Read) | GET /api/items |

|  |  |  |
| --- | --- | --- |
| **POST** | Create new data | POST /api/items |

|  |  |  |
| --- | --- | --- |
| **PUT** | Update/Replace existing data | PUT /api/items/1 |

|  |  |  |
| --- | --- | --- |
| **PATCH** | Update/Modify partial data | PATCH /api/items/1 |

|  |  |  |
| --- | --- | --- |
| **DELETE** | Remove data | DELETE /api/items/1 |

**3. REST API Principles**

1. **Stateless**: Each request from a client contains all the information needed to process it. The server does not store session state.
2. **Client-Server Separation**: The client and server are independent of each other.
3. **Uniform Interface**: Resources are accessed using URIs and standard HTTP methods.
4. **Layered System**: A client cannot tell if it is communicating directly with the server or an intermediary.
5. **. REST API Response Status Codes**

| **Code** | **Meaning** | **Description** |
| --- | --- | --- |
| **200** | OK | Request succeeded (GET, PUT, PATCH). |
| **201** | Created | Resource created successfully (POST). |
| **204** | No Content | Request succeeded, no content to return. |
| **400** | Bad Request | Invalid input from client. |
| **401** | Unauthorized | Authentication required. |
| **403** | Forbidden | Access to the resource is denied. |
| **404** | Not Found | Resource does not exist. |
| **500** | Internal Server Error | Server encountered an unexpected condition. |

**Authentication in REST API**

* **API Key**: A unique key for accessing the API.
* **OAuth**: Token-based authentication.
* **JWT (JSON Web Tokens)**: Secure and stateless token for API authentication.

| **Feature** | **REST** | **SOAP** |
| --- | --- | --- |

|  |  |  |
| --- | --- | --- |
| **Protocol** | HTTP | HTTP, SMTP, TCP |

|  |  |  |
| --- | --- | --- |
| **Data Format** | JSON, XML | XML |

|  |  |  |
| --- | --- | --- |
| **Flexibility** | More flexible | More rigid |

|  |  |  |
| --- | --- | --- |
| **Ease of Use** | Easier to implement | More complex |

**What is Pytest?**

* **Pytest** is a testing framework for Python.
* It allows for simple unit testing and supports advanced features like fixtures, parameterized tests, and plugins.
* Works seamlessly with other testing tools like unittest.

**2. Advantages of Pytest**

1. Simple and easy-to-use syntax.
2. Automatic test discovery.
3. Supports fixtures for setup and teardown logic.
4. Supports parameterized testing.
5. Rich ecosystem of plugins (e.g., pytest-django, pytest-cov for coverage).
6. Integrates with CI/CD pipelines.

**3. Writing a Basic Test**

1. Test functions must start with test\_ or class names must start with Test.
2. Assertions are written using Python's assert keyword.

**Example:**

# test\_sample.py

def test\_addition():

assert 1 + 1 == 2

def test\_subtraction():

assert 5 - 3 == 2

Run the tests:

pytest test\_sample.py

**4. Fixtures**

* Fixtures provide a way to manage setup and teardown logic for tests.
* Use the @pytest.fixture decorator.

**Example:**

import pytest

@pytest.fixture

def sample\_data():

return [1, 2, 3]

def test\_sum(sample\_data):

assert sum(sample\_data) == 6

**5. Parameterized Tests**

* Test the same logic with multiple sets of input data using @pytest.mark.parametrize.

**Example:**

python

Copy code

import pytest

@pytest.mark.parametrize("x, y, expected", [

(1, 2, 3),

(2, 3, 5),

(3, 5, 8)

])

def test\_addition(x, y, expected):

assert x + y == expected

**6. Skipping Tests**

* Skip tests using @pytest.mark.skip or conditionally using @pytest.mark.skipif.

**Example:**

import pytest

@pytest.mark.skip(reason="This test is not ready yet.")

def test\_incomplete():

assert False

@pytest.mark.skipif(3 > 2, reason="Conditionally skipped.")

def test\_conditional():

assert True

**7. Expected Failures**

* Mark tests as expected failures using @pytest.mark.xfail.

**Example:**

import pytest

@pytest.mark.xfail

def test\_known\_issue():

assert 1 == 2 # This test will be marked as "XFAIL"

**8. Running Tests**

* Run all tests:

pytest

* Run tests from a specific file:

pytest test\_file.py

* Run a specific test function:

pytest test\_file.py::test\_function

**9. Test Coverage**

* Install pytest-cov:

pip install pytest-cov

* Run tests with coverage:

pytest --cov=your\_module tests/

**10. Pytest Assertions**

Pytest supports Python's built-in assert keyword with enhanced error messages.

python

Copy code

def test\_example():

assert 3 > 2

assert "hello" in "hello world"

assert [1, 2] == [1, 2]

**11. Useful Pytest Commands**

| **Command** | **Description** |
| --- | --- |
| pytest -v | Verbose mode: Show detailed output. |
| pytest -x | Stop after the first failure. |
| pytest --maxfail=2 | Stop after two failures. |
| pytest -k "keyword" | Run tests matching the keyword. |
| pytest -m "marker" | Run tests marked with a specific marker. |
| pytest --tb=short | Display shorter traceback on failure. |

**12. Markers**

Markers are used to group or categorize tests. You can define custom markers.

**Example:**python

import pytest

@pytest.mark.slow

def test\_long\_running():

assert True

Run tests with a specific marker:

pytest -m "slow"

**13. Plugins**

Popular Pytest plugins:

1. **pytest-django**: For testing Django applications.
2. **pytest-cov**: For measuring test coverage.
3. **pytest-mock**: For mocking in tests.
4. **pytest-html**: For generating HTML reports.

Install plugins using pip:

pip install pytest-django pytest-cov

**BDD Framework: Key Notes**

**1. What is BDD?**

* **BDD** stands for **Behavior-Driven Development**.
* It is a software development approach that encourages collaboration between developers, testers, and non-technical stakeholders.
* Focuses on **defining application behavior** using plain, human-readable language.

**2. Key Components of BDD**

1. **Gherkin Language**:
   * A plain-text language used to write scenarios.
   * Follows a **Given-When-Then** structure to describe behavior.

Example:

gherkin

Copy code

Feature: Login functionality

Scenario: Successful login

Given the user is on the login page

When the user enters valid credentials

Then the user should be redirected to the dashboard

1. **Features and Scenarios**:
   * **Feature**: Describes a specific functionality of the application.
   * **Scenario**: Describes a specific example or behavior of the feature.
2. **Step Definitions**:
   * Python (or other language) methods that map to Gherkin steps.

**3. Tools for BDD**

| **Language** | **BDD Tool** |
| --- | --- |
| Python | Behave, pytest-bdd |
| Java | Cucumber, JBehave |
| JavaScript | Cucumber.js |
| Ruby | Cucumber |

**4. Writing BDD Tests with Behave (Python Example)**

1. **Install Behave**:

bash

Copy code

pip install behave

1. **Project Structure**:

markdown

Copy code

features/

login.feature

steps/

login\_steps.py

1. **Feature File (login.feature)**:

gherkin

Copy code

Feature: Login functionality

Scenario: Successful login

Given the user is on the login page

When the user enters valid credentials

Then the user should see the dashboard

1. **Step Definitions (login\_steps.py)**:

python

Copy code

from behave import given, when, then

@given("the user is on the login page")

def step\_given\_user\_on\_login\_page(context):

context.page = "login"

@when("the user enters valid credentials")

def step\_when\_user\_enters\_credentials(context):

context.logged\_in = True

@then("the user should see the dashboard")

def step\_then\_user\_sees\_dashboard(context):

assert context.logged\_in is True

1. **Run Tests**:

bash

Copy code

behave

**5. Advantages of BDD**

1. Encourages collaboration between technical and non-technical stakeholders.
2. Ensures a shared understanding of requirements.
3. Scenarios act as **living documentation** for the application.
4. Improves test coverage by focusing on user behavior.

**6. Gherkin Keywords**

| **Keyword** | **Description** |
| --- | --- |
| **Feature** | Defines a feature or functionality. |
| **Scenario** | Describes a specific behavior or example. |
| **Given** | Describes the initial context or precondition. |
| **When** | Describes the action or event. |
| **Then** | Describes the expected outcome. |
| **And**/**But** | Additional steps to extend scenarios. |

**7. BDD vs TDD**

| **Aspect** | **BDD** | **TDD** |
| --- | --- | --- |
| **Focus** | Behavior of the application | Unit tests for individual components |
| **Language** | Plain English (Gherkin) + Code | Pure code |
| **Stakeholder Involvement** | High (non-technical stakeholders) | Low (primarily developers) |

{

"end\_year": "",

"intensity": 6,

"sector": "Energy",

"topic": "gas",

"insight": "Annual Energy Outlook",

"url": "http://www.eia.gov/outlooks/aeo/pdf/0383(2017).pdf",

"region": "Northern America",

"start\_year": "",

"impact": "",

"added": "January, 20 2017 03:51:25",

"published": "January, 09 2017 00:00:00",

"country": "United States of America",

"relevance": 2,

"pestle": "Industries",

"source": "EIA",

"title": "U.S. natural gas consumption is expected to increase during much of the projection period.",

"likelihood": 3

},

**wap that ask the user input th integer array in size pass argument array to the function and the function will check the array n fn display the prime number**

**Timeline and Relationship:**

1. **BCPL** (1966):
   * Developed by **Martin Richards** as a typeless, minimalist language intended for system programming and compiler writing.
   * BCPL influenced Ken Thompson's work on B.
2. **B** (1970):
   * Created by **Ken Thompson** at Bell Labs.
   * B was essentially a stripped-down version of BCPL, tailored for the PDP-7 minicomputer, which had limited memory.
   * B retained the typeless nature of BCPL and introduced some syntactic changes.
3. **C** (1972):
   * Developed by **Dennis Ritchie**, also at Bell Labs, as an evolution of B.
   * C added data types and other features, making it more robust for systems programming, including writing the UNIX operating system.

**Summary:**

* **BCPL**: Created by Martin Richards.
* **B**: Derived from BCPL by Ken Thompson.
* **C**: Built on B by Dennis Ritchie.

 **Early Development (1969–1972)**:

* C was developed by **Dennis Ritchie** at **Bell Labs**, primarily to rewrite the UNIX operating system, making it portable and efficient.
* The language evolved from **B**, which itself was derived from **BCPL**.

 **Internal Use (1972–1978)**:

* C was initially used internally at Bell Labs for developing the UNIX operating system and related tools.
* It gained popularity among researchers and developers within AT&T and universities.

 **Public Release (1978)**:

* The publication of **"The C Programming Language"** by Kernighan and Ritchie made the language widely accessible.
* This book documented what is now called **K&R C**, the first standardized version of the language.

 **Standardization (1989)**:

* The American National Standards Institute (ANSI) standardized C in **1989**, resulting in **ANSI C (C89)**, which included improvements and formalized the language specification.
* The International Organization for Standardization (ISO) adopted it in **1990** as **ISO C (C90)**.

**Definition: Constants are fixed values that cannot be changed during program execution.**

**Operators**

* **Definition**: Operators perform operations on variables and values.

| **Type** | **Operators** |
| --- | --- |
| Arithmetic | +, -, \*, /, % |
| Relational | ==, !=, <, > |
| Logical | &&, ` |
| Assignment | =, +=, -= |
| Bitwise | &, ` |

**Definition**: Data types specify the type of data a variable can store, such as integers, floating-point numbers, or characters.

| **Data Type** | **Size** | | **Example** |
| --- | --- | --- | --- |
| char | 1 byte | | 'A', 'Z' |
| int | 4 bytes | | 10, -20 |
| float | 4 bytes | | 3.14, -0.001 |
| double | 8 bytes | | 3.14159265 |
| void | No size | | Used for no value |
| **Token Type** | | **Example** | | |
| **Keyword** | | int, return, if | | |
| **Identifier** | | myVariable, sum, main | | |
| **Constant** | | 100, 3.14, 'A', "Hello" | | |
| **Operator** | | +, ==, &&, = | | |
| **Special Symbol** | | {}, [], (), ;, # | | |
| **String** | | "Hello, World!" | | |

| **Storage Class** | **Scope** | **Lifetime** | **Memory** | **Keyword** |
| --- | --- | --- | --- | --- |
| auto | Block (local) | Block | Stack | auto |
| register | Block (local) | Block | CPU Register | register |
| static | Block/File | Entire program | Data Segment | static |
| extern | Global (external) | Entire program | Data Segment | extern |
| \_Thread\_local | Block/Thread | Thread lifetime | Data Segment | \_Thread\_local |

**Call by Value**

* **Definition**: In Call by Value, a copy of the actual argument is passed to the function. Any modifications made to the parameter inside the function do not affect the original variable.
* **Key Characteristics**:
  + The function works on a copy of the argument.
  + The original variable remains unchanged.
  + Suitable when you do not want the function to modify the original variable.

**Call by Reference**

* **Definition**: In Call by Reference, the address (reference) of the actual argument is passed to the function. The function works directly on the original variable, so changes are reflected in the original variable.
* **Key Characteristics**:
  + The function works on the actual argument through its address.
  + The original variable can be modified by the function.
  + Typically achieved in C by using pointers.

**Actual Argument**

* These are the **values or variables** provided in a function call.
* They exist in the calling function and are passed to the called function.
* Actual arguments are evaluated and copied (for call by value) or their address is passed (for call by reference).

**Formal Argument**

* These are the **parameters** defined in the function declaration or definition.
* They act as placeholders and receive the values of actual arguments during the function call.
* They exist only within the scope of the function.

A **recursion function** is a function that calls itself either directly or indirectly to solve a smaller subproblem of the original problem. Recursion is typically used to solve problems that can be divided into similar subproblems.

**Arrays**

* An array is a collection of elements of the same data type.
* The array name itself acts as a pointer to the first element of the array.
* Elements in the array are accessed using an index.

**Pointers**

* A pointer is a variable that stores the memory address of another variable.
* Pointers can also be used to access elements of an array by arithmetic on the pointer.

A **dangling pointer** is a pointer that points to a memory location that has already been **freed or deallocated**. Accessing a dangling pointer can lead to undefined behavior, such as reading incorrect or random values from memory.

| **Aspect** | **Array** | **Pointer** | **Void Pointer** |
| --- | --- | --- | --- |

|  |  |  |  |
| --- | --- | --- | --- |
| **Definition** | Collection of elements of the same type. | Variable that stores the address of another variable. | Pointer that can store the address of any type. |

|  |  |  |  |
| --- | --- | --- | --- |
| **Memory** | Contiguous block allocated at compile time. | Points to a memory location, can be reassigned. | Points to any data type but requires casting. |

|  |  |  |  |
| --- | --- | --- | --- |
| **Access** | Access using index (arr[i]). | Dereferencing (\*ptr). | Requires type casting to dereference. |

|  |  |  |  |
| --- | --- | --- | --- |
| **Flexibility** | Fixed size and type. | Flexible for type and address changes. | Highly flexible, can point to any type. |

|  |  |  |  |
| --- | --- | --- | --- |
| **Usage** | For handling fixed-size data structures. | For dynamic memory and indirect manipulation. | For generic programming and data-type agnostic operations. |

|  |  |  |  |
| --- | --- | --- | --- |
| **Arithmetic** | Supports pointer arithmetic implicitly (arr + i). | Supports pointer arithmetic. | Cannot perform arithmetic directly. |

 **Memory Initialization**:

* malloc allocates uninitialized memory.
* calloc allocates and initializes the memory to zero.

 **Resizing**:

* malloc cannot resize; realloc is used to resize the already allocated memory.

 **Purpose**:

* malloc is used when you need a block of memory without initialization.
* calloc is preferred when you want the allocated memory to be zero-initialized.
* realloc is used to change the size of a previously allocated memory block.

 **Overhead**:

* calloc introduces extra overhead due to initialization.
* realloc may incur additional overhead if the block needs resizing.

**free(ptr)**:

* **Purpose**: Frees the dynamically allocated memory, making it available for reuse.
* **Usage**:

c

Copy code

free(ptr); // Free the allocated memory

ptr = NULL; // Set pointer to NULL after freeing

* **Importance**: Failure to free memory results in memory leaks, where memory allocated remains inaccessible, reducing efficiency.

Command line arguments allow you to pass input data to a C program when running it from the terminal. These arguments are passed as parameters to the main function.  
  
The preprocessor is a built-in feature that processes directives (instructions) before the actual compilation of the code. It modifies the source code by including header files, defining macros, and performing conditional compilation.

**Common Preprocessor Directives:**

1. **#include**: Includes a header file.
2. **#define**: Defines a macro.
3. **#undef**: Undefines a macro.
4. **#if, #else, #endif**: Conditional compilation.
5. **#ifdef, #ifndef**: Checks if a macro is defined.
6. **#pragma**: Provides compiler-specific directives.

A **macro** is a preprocessor directive that defines symbolic constants or functions, replacing the name with the value before compilation.

A **2D array** is an array of arrays, used to represent matrices or tables

**4. String Handling Functions in C:**

C provides several built-in functions to handle strings (arrays of characters).

**Common String Handling Functions:**

1. **strlen(s)**: Returns the length of the string.
2. **strcpy(dest, src)**: Copies a string from src to dest.
3. **strcat(dest, src)**: Concatenates src to the end of dest.
4. **strcmp(s1, s2)**: Compares two strings lexicographically.
5. **strncmp(s1, s2, n)**: Compares the first n characters of two strings.
6. **strchr(s, c)**: Finds the first occurrence of a character c in the string.
7. **strrchr(s, c)**: Finds the last occurrence of a character c in the string.
8. **strstr(haystack, needle)**: Finds the first occurrence of needle in haystack.
9. **strtok(s, delim)**: Splits a string into tokens based on a delimiter.

**Preprocessing:**

* **The preprocessor (cpp) scans the source code.**
* **It handles directives like #include, #define, #ifdef, etc.**
* **Removes comments and replaces macros with their values**

**Compilation:**

* **The preprocessed code is passed to the compiler (gcc, cc, etc.).**
* **It translates the C source code into assembly language (object code).**
* **During this phase, syntax errors are checked.**

**Assembly:**

* The compiler generates assembly code from the object file.
* Assembly is a low-level representation of the program.

**Linking:**

* The **linker (ld, gcc -o)** combines object files and other necessary libraries (e.g., libc).
* Resolves references to external functions (like printf from stdio.h).
* Produces the final executable file.

**Execution:**

* The operating system loads the executable into memory.
* Control is passed to the main function.
* The program starts executing.

Source Code (.c) --> Preprocessing --> Compiling --> Assembly --> Linking --> Executable (.out or .exe)

| | | | |

| | | | v

v v v v Executable Program