**Practical: 4**

**Aim: Explore enum.**

**Code:**

#[allow(dead\_code)]

enum Payment {

Cash(f64),

Card(String, u8),

Crypto { currency: String, amount: f64 },

UPI(UpiData),

}

struct UpiData {

upi\_id: String,

phone\_number: String,

}

fn main() {

let payment = Payment::Card(String::from("Utsav Balar"), 12);

match payment {

Payment::Cash(amount) => println!("Cash Payment: {}", amount),

Payment::Card(name, amount) => {

println!("Card Owner: {}, Card Payment: {}", name, amount)

}

Payment::Crypto { currency, amount } => {

println!("Crypto Currency: {}, Amount: {}", currency, amount)

}

Payment::UPI(data) => println!(

"UPI ID: {}, Phone Number: {}",

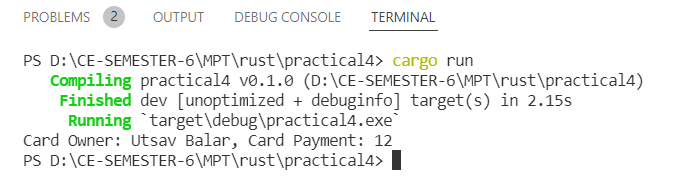
data.upi\_id, data.phone\_number

),

}

}

**Output:**

****

let payment = Payment::Crypto {

currency: String::from("Bitcoin"),

amount: 0.00000012,

};

Roman symbols using enum

**main.rs**

mod roman\_symbols;

use roman\_symbols::RomanSymbols;

fn main() {

use std::io;

let mut input = String::new();

println!("Enter a number: ");

io::stdin()

.read\_line(&mut input)

.expect("Failed to read line");

let input = input.trim().parse().expect("Failed to parse input");

if let Some(num) = RomanSymbols::int\_to\_roman(input) {

println!("{}", num);

if let Some(num) = RomanSymbols::roman\_to\_int(&num) {

println!("{}", num);

} else {

println!("Failed to convert to int");

}

} else {

println!("Invalid value");

}

}

**roman\_symbols.rs**

pub static MAX: u16 = 3999;

#[derive(Debug)]

pub enum RomanSymbols {

I,

IV,

V,

IX,

X,

XL,

L,

XC,

C,

CD,

D,

CM,

M,

}

impl RomanSymbols {

pub fn value(&self) -> u16 {

match self {

RomanSymbols::I => 1,

RomanSymbols::IV => 4,

RomanSymbols::V => 5,

RomanSymbols::IX => 9,

RomanSymbols::X => 10,

RomanSymbols::XL => 40,

RomanSymbols::L => 50,

RomanSymbols::XC => 90,

RomanSymbols::C => 100,

RomanSymbols::CD => 400,

RomanSymbols::D => 500,

RomanSymbols::CM => 900,

RomanSymbols::M => 1000,

}

}

pub fn from\_str(value: &str) -> RomanSymbols {

match value {

"I" => RomanSymbols::I,

"IV" => RomanSymbols::IV,

"V" => RomanSymbols::V,

"IX" => RomanSymbols::IX,

"X" => RomanSymbols::X,

"XL" => RomanSymbols::XL,

"L" => RomanSymbols::L,

"XC" => RomanSymbols::XC,

"C" => RomanSymbols::C,

"CD" => RomanSymbols::CD,

"D" => RomanSymbols::D,

"CM" => RomanSymbols::CM,

"M" => RomanSymbols::M,

\_ => panic!("Invalid value, {:?}", value),

}

}

pub fn symbols() -> Vec<String> {

vec![

"I".to\_string(),

"IV".to\_string(),

"V".to\_string(),

"IX".to\_string(),

"X".to\_string(),

"XL".to\_string(),

"L".to\_string(),

"XC".to\_string(),

"C".to\_string(),

"CD".to\_string(),

"D".to\_string(),

"CM".to\_string(),

"M".to\_string(),

]

}

pub fn values() -> Vec<u16> {

vec![1, 4, 5, 9, 10, 40, 50, 90, 100, 400, 500, 900, 1000]

}

pub fn roman\_to\_int(value: &String) -> Option<u16> {

let (mut n, mut result) = (0, 0);

for i in value.chars().rev() {

let v = RomanSymbols::from\_str(&i.to\_string());

if v.value() >= n {

result += v.value();

} else {

result -= v.value();

}

n = v.value();

}

if result > MAX {

None

} else {

Some(result)

}

}

pub fn roman\_pairs() -> Vec<(String, u16)> {

let symbols = RomanSymbols::symbols();

let values = RomanSymbols::values();

let mut result = Vec::new();

for (i, symbol) in symbols.iter().enumerate() {

result.push((symbol.to\_string(), values[i]));

}

result.reverse();

result

}

pub fn int\_to\_roman(n: u16) -> Option<String> {

if n <= 0 || n > MAX {

panic!("Argument must be between 1 and 3999");

}

let mut result = String::new();

let mut n = n;

for (symbol, value) in RomanSymbols::roman\_pairs() {

while n >= value {

n -= value;

result.push\_str(symbol.as\_str());

}

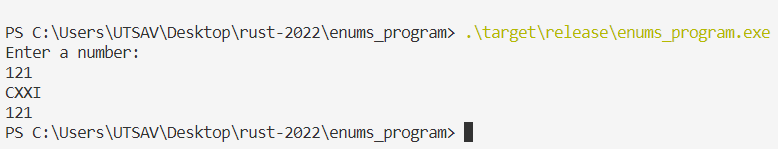
}

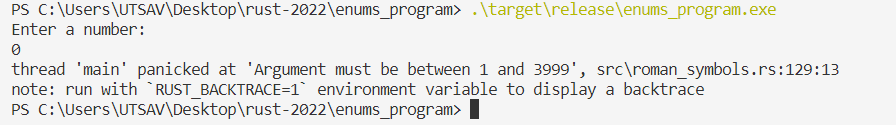
Some(result)

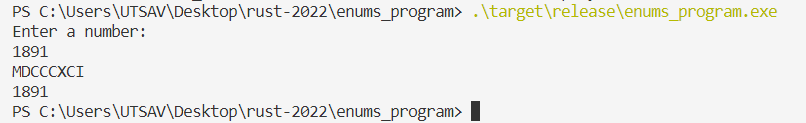
}

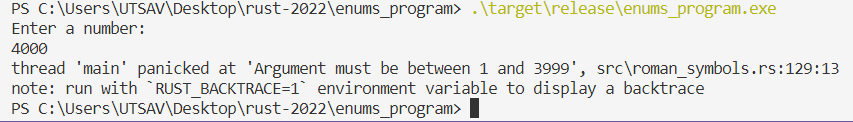
}

**Output:**









**Practical: 5**

**Aim: Explore packages and crates**

**Code:**

**Cargo.toml**

[package]

name = "modules\_test"

version = "0.1.0"

edition = "2021"

# See more keys and their definitions at https://doc.rust-lang.org/cargo/reference/manifest.html

[lib]

name = "creature\_library"

path = "./src/creature\_library/lib.rs"

[dependencies]

rand\_derive2 = "0.1.17"

rand = "0.8.5"

**bin/main.rs**

#[allow(unused\_imports)]

use creature\_library::creature::{Creature, Body};

use creature\_library::get\_type\_creature::TypeToStr;

fn main() {

// let creature1 = Creature::Dog(Body {

// eyes: 2,

// nose: 1,

// hands: 0,

// legs: 4,

// mouth: 1,

// });

// let creature2 = Creature::Cat(Body {

// eyes: 2,

// nose: 1,

// hands: 0,

// legs: 4,

// mouth: 1,

// });

// println!("Creature 1 => {}", creature1.get\_type\_creature());

// println!("Creature 2 => {}", creature2.get\_type\_creature());

// creature1.creature\_exists();

// creature2.creature\_exists();

// let creature3 = Creature::new\_creature(&Creature::Human(Body {

// hands: 2,

// legs: 2,

// mouth: 1,

// nose: 1,

// eyes: 2,

// }));

// println!("Creature 3 => {}", creature3.get\_type\_creature());

// let creature4 = Creature::copy\_creature(&creature3);

// println!("Creature 4 => {}", creature4.get\_type\_creature());

// creature3.creature\_exists();

// creature4.creature\_exists();

// let creature5 = Creature::new\_creature(&Creature::MythicalBeast(Body {

// hands: 8,

// legs: 8,

// mouth: 4,

// nose: 4,

// eyes: 8,

// }));

// creature5.creature\_exists();

use std::io;

let your\_creature: Creature = Creature::generate\_random\_creature();

let mut input = String::new();

println!("Enter your name: ");

io::stdin().read\_line(&mut input).expect("Failed to read line");

println!("{} is the {}", input.trim(), your\_creature.get\_type\_creature());

}

**creature\_library/creature.rs**

// crate to generate random creature

use rand\_derive2::RandGen;

// crate that implements method to get type of creature

use crate::get\_type\_creature::TypeToStr;

#[derive(RandGen)]

#[allow(dead\_code)]

#[derive(Debug)]

pub struct Body {

pub eyes: u8,

pub nose: u8,

pub hands: u8,

pub legs: u8,

pub mouth: u8,

}

#[derive(RandGen)]

#[allow(dead\_code)]

#[derive(Debug)]

pub enum Creature {

Dog(Body),

Cat(Body),

Human(Body),

MythicalBeast(Body),

Donkey(Body),

Monkey(Body),

Snake(Body),

Dragon(Body),

Fish(Body),

Bird(Body),

Lizard(Body),

Frog(Body),

Bat(Body),

Bear(Body),

Elephant(Body),

Rhino(Body),

Giraffe(Body),

}

impl Creature {

pub fn check\_creature(&self) -> Option<bool> {

match self {

Creature::Dog(\_)

| Creature::Cat(\_)

| Creature::Human(\_)

| Creature::MythicalBeast(\_)

| Creature::Donkey(\_)

| Creature::Monkey(\_)

| Creature::Snake(\_)

| Creature::Dragon(\_)

| Creature::Fish(\_)

| Creature::Bird(\_)

| Creature::Lizard(\_)

| Creature::Frog(\_)

| Creature::Bat(\_)

| Creature::Bear(\_)

| Creature::Elephant(\_)

| Creature::Rhino(\_)

| Creature::Giraffe(\_) => Some(true),

}

}

pub fn new\_creature(&self) -> &Self {

self

}

pub fn copy\_creature(&self) -> &Self {

Self::new\_creature(&self)

}

pub fn generate\_random\_creature() -> Self {

Self::generate\_random()

}

pub fn creature\_exists(&self) {

match self.check\_creature() {

Some(thing) => {

if thing == true {

println!("{} exists!", self.get\_type\_creature())

}

}

\_ => {

println!("{} Doesn't exist", self.get\_type\_creature())

}

}

}

}

**creature\_library/get\_creature\_type.rs**

use crate::creature::Creature;

pub trait TypeToStr {

fn get\_type\_creature(&self) -> &'static str;

}

impl TypeToStr for Creature {

fn get\_type\_creature(&self) -> &'static str {

match self {

Creature::Dog(\_) => "Dog",

Creature::Cat(\_) => "Cat",

Creature::Human(\_) => "Human",

Creature::MythicalBeast(\_) => "MythicalBeast",

Creature::Donkey(\_) => "Donkey",

Creature::Monkey(\_) => "Monkey",

Creature::Snake(\_) => "Snake",

Creature::Dragon(\_) => "Dragon",

Creature::Fish(\_) => "Fish",

Creature::Bird(\_) => "Bird",

Creature::Lizard(\_) => "Lizard",

Creature::Frog(\_) => "Frog",

Creature::Bat(\_) => "Bat",

Creature::Bear(\_) => "Bear",

Creature::Elephant(\_) => "Elephant",

Creature::Rhino(\_) => "Rhino",

Creature::Giraffe(\_) => "Giraffe",

}

}

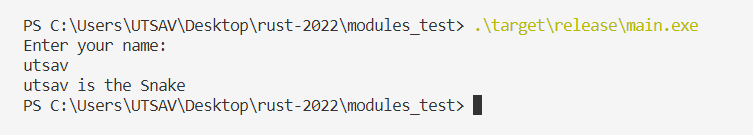
}

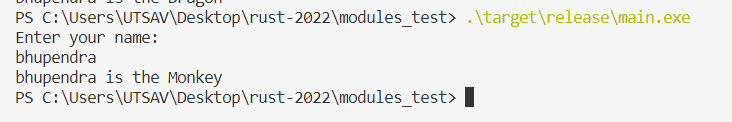
**creature\_library/lib.rs**

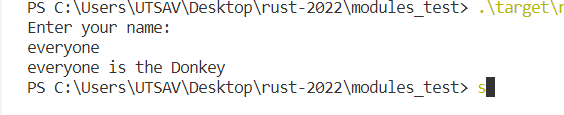
pub mod creature;

pub mod get\_type\_creature;

**Output:**

****

****

****

**Practical: 6**

**Aim: Explore keywords and globe operator**

**Code:**

**main.rs**

// Using globe operator to import public methods from operations

use operations::\*;

// Using io crate to read input from stdin

use std::io;

fn main() {

let mut input = String::new();

let mut operation = String::new();

println!("Enter operation to perform: \n+ => Add\n- => Subtract\n\* => Multiply\n/ => Divide\ns => Square\nc => Cube");

io::stdin().read\_line(&mut operation).unwrap();

// Extracting the first character of the operation string

let operation: char = operation.trim().chars().nth(0).unwrap();

// Checking if the operation is valid

match operation {

'+' => {

println!("Enter two numbers seperated by space: ");

io::stdin().read\_line(&mut input).unwrap();

let input: Vec<&str> = input.trim().split(' ').collect();

let n = input[0].parse::<i32>().unwrap();

let m = input[1].parse::<i32>().unwrap();

println!("{} + {} = {}", n, m, add::add(n, m));

}

'-' => {

println!("Enter two numbers seperated by space: ");

io::stdin().read\_line(&mut input).unwrap();

let input: Vec<&str> = input.trim().split(' ').collect();

let n = input[0].parse::<i32>().unwrap();

let m = input[1].parse::<i32>().unwrap();

println!("{} - {} = {}", n, m, subtract::subtract(n, m));

}

'\*' => {

println!("Enter two numbers seperated by space: ");

io::stdin().read\_line(&mut input).unwrap();

let input: Vec<&str> = input.trim().split(' ').collect();

let n = input[0].parse::<i32>().unwrap();

let m = input[1].parse::<i32>().unwrap();

println!("{} \* {} = {}", n, m, multiply::multiply(n, m));

}

'/' => {

println!("Enter two numbers seperated by space: ");

io::stdin().read\_line(&mut input).unwrap();

let input: Vec<&str> = input.trim().split(' ').collect();

let n = input[0].parse::<i32>().unwrap();

let m = input[1].parse::<i32>().unwrap();

println!("{} / {} = {}", n, m, divide::divide(n, m));

}

's' => {

println!("Enter a number: ");

io::stdin().read\_line(&mut input).unwrap();

let input: i32 = input.trim().parse().unwrap();

println!("{}^2 = {}", input, square::square(input));

}

'c' => {

println!("Enter a number: ");

io::stdin().read\_line(&mut input).unwrap();

let input: i32 = input.trim().parse().unwrap();

println!("{}^3 = {}", input, cube::cube(input));

}

\_ => println!("{}", "Invalid operation"),

}

// Addition of two string with add but changing function name

use operations::str\_add as StrAdd;

println!("Addition of two strings {}", StrAdd::add("Hello ", "World!"));

}

**add.rs**

pub fn add<T>(n1: T, n2: T) -> T

where

T: std::ops::Add<Output = T>,

{

n1 + n2

}

**cube.rs**

pub fn cube<T>(n: T) -> T

where

T: Copy + std::ops::Mul<Output = T>,

{

n \* n \* n

}

**divide.rs**

pub fn divide<T>(n1: T, n2: T) -> T

where

T: std::ops::Div<Output = T>,

{

n1 / n2

}

**multiply.rs**

pub fn multiply<T>(n1: T, n2: T) -> T

where

T: std::ops::Mul<Output = T>,

{

n1 \* n2

}

**square.rs**

pub fn square<T>(n: T) -> T

where

T: Copy + std::ops::Mul<Output = T>,

{

n \* n

}

**str\_add.rs**

pub fn add(str1: &str, str2: &str) -> String {

format!("{}{}", str1, str2)

}

**subtract.rs**

pub fn subtract<T>(n1: T, n2: T) -> T

where

T: std::ops::Sub<Output = T>,

{

n1 - n2

}

**lib.rs**

pub mod add;

pub mod cube;

pub mod divide;

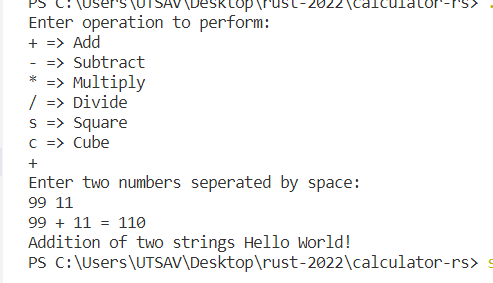
pub mod multiply;

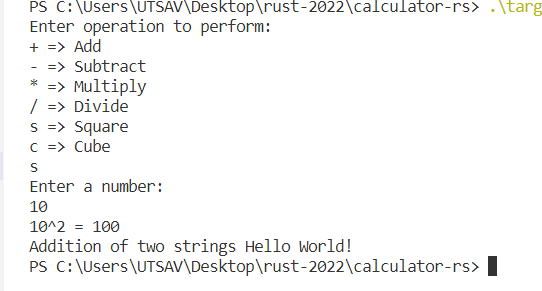
pub mod square;

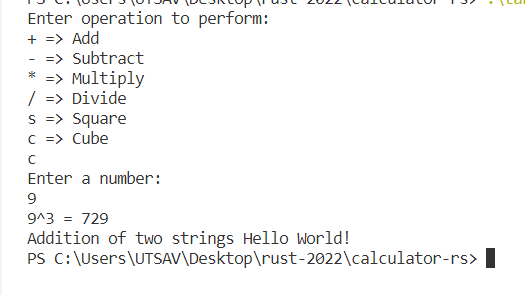
pub mod subtract;

pub mod str\_add;

**Output:**

****

****

****