//strust

//结构体,自定义数据类型

/\*

struct User{

  username:String,

  email:String,

  sign\_in\_count:u64,

  sign\_in\_count:u64,

  active:boll,

}\*/

//需要创建struct实例,为每个字段制定具体值,无需按照声明时顺序指定

// struct User{

//   username:String,

//   email:String,

//   sign\_in\_count:u64,

//   active:bool,

// }

// fn main(){

//   println!("hello");

//   let user1 = User{

//       email:String::from("123687"),

//       username:String::from("87462"),

//       active:true,

//       sign\_in\_count:1,

//   };

// }

//在struct里取值,使用点标记法

//需要结构体实例可变

//user1.email = String::from("14234312");

//一旦实例可变,则实例所有字段可变

//可以作为函数返回值

// fn build\_user(email:String,username:String) -> User{

//   User{

//     email:email,

//     username:uername,

//   }

// }

//字段值和值对应的的变量名相同时可以简写即email:email,变成email

//struct更新语法

//基于某一现有struct实例来创建一个新的实例时,新实例一些值和旧的相同

//let user1 = User{

//       email:String::from("123687"),

//       username:String::from("87462"),

//       active:true,

//       sign\_in\_count:1,

//   };

//Tuple struct

//可以自定义的类似tuple的struct,整体有名,但元素没有名

//struct Color(i32,i32,i32);let black = Color(0,0,0);

//Unit-Like Struct

//可以定义没有任何字段的struct,叫Unit-Like struct

//适用于需要在某个类型上实现某个trait(当作接口),但有没有想要储存的数据

//struct数据所有权

// struct User{

//   username:String,

//   email:String,

//   sign\_in\_count:u64,

//   active:bool,

// }

//这里的字段使用的是string而不是&str,所以该实例拥有所有数据,只要struct实例有效,那么字段数据也是有效的

//struct里可以存放引用,但这需要使用生命周期,生命周期保证只要实例有效,那么引用也是有效的,如果不使用生命周期,会报错

//例题:计算长方形面积

// #[derive(Debug)]

// struct Rectangle{

//   width:u32,

//   length:u32,

// }

// fn main(){

//                   // let mut w:u32 = 5;

//                   // let mut l:u32 = 3;

//                   // println!("{}",area(w,l));

//         // let rect = (3,5);

//         // println!("{}",area(rect));

//   let rect = Rectangle{

//     width: 3,

//     length: 5,

//   };

//   println!("{}",area(&rect));

//   println!("{:#?}",rect);

// }

//                     // fn area(width:u32,length:u32) -> u32{

//                     //   width \* length

//                     // }

//         // fn area(dim: (u32,u32)) -> u32{

//         //   dim.0 \* dim.1

//         // }

// fn area(rect:&Rectangle) -> u32{

//   rect.width \* rect.length

// }

//格式化方法

//std::fmt::Display

//std::fmt::Debug

//#[derive(Debug)]

//{:?}(较简单)

//{:#?}(较清晰)

//struct的方法

//和函数类似有fn关键字,名称,参数,返回值

//方法在struct(或enum,trait对象)上下文中定义,方法第一个参数是self,表示方法被调用的struct实例

//impl块中定义,第一个参数可以是&self,也可以获得所有权或可变借用

//有更好的组织

//rust会自动引用或解引用,调用方法时会发生这种行为

//在调用方法时,rust会自动添加&,&mut或\*,以便object可以匹配方法签名

//以下两种代码效果相同

// p1.distance(&p2);

// (&p1).distance(&p2);

// #[derive(Debug)]

// struct Rectangle{

//   width:u32,

//   length:u32,

// }

// impl Rectangle{

//   fn area(&self) -> u32{

//     self.width \* self.length

//   }

//   fn can\_hold(&self,other:&Rectangle) -> bool{

//     self.width > other.width && self.length > other.length

//   }

//   fn square(size: u32) -> Rectangle{//当需要调用的参数是struct实例,用self.如果第一个是一个静态参数,那么第二个参数不能再用self了(关联函数不包含self)

//     Rectangle{

//       width: size,

//       length: size,

//     }

//   }

// }

// fn main(){

//   let s = Rectangle::square(30);//关联函数

//   let rect1 = Rectangle{

//     width:30,

//     length:50,

//   };

//   let rect2 = Rectangle{

//     width:10,

//     length:40,

//   };

//   let rect3 = Rectangle{

//     width:35,

//     length:55,

//   };

//   // println!("{}",rect.area());//实例.方法名调用方法,不需要显式调用参数,因为实力本身就是self

//   println!("{}",rect1.can\_hold(&rect2));

//   println!("{}",rect1.can\_hold(&rect3));

//   println!("{:#?}",rect1);

// }

//关联函数

//可以在impl块里定义的不把self作为第一个参数的函数(不是方法)

//String::from()

//关联函数通常用于构造器,也就是说通常被用于创建关联类型的一个实例

//::符号,用于关联函数,也可以用于模块创建的命名空间

//每个struct允许有多个impl块

//枚举与模式匹配

//枚举允许我们列举所有可能指来定义类型

//枚举所有可能指叫枚举的变体,枚举变体都位于标识符命名空间下使用::分隔

// enum IpAddrKind{

//   v4,

//   v6,

// }

// // let four = IpAddrKind::v4;

// // let six = IpAddrKind::v6;

// // fn main(){

// //   let four = IpAddrKind::v4;

// //   let six = IpAddrKind::v6;

// // }

// fn route(ip\_kind: IpAddrKind){}

// //将数据附加到枚举变体中

// struct IpAddr{

//   kind: IpAddrKind,

//   adress: String,

// }

// fn main(){

//   let home = IpAddr{

//     kind:IpAddrKind::v4,

//     adress:String::from("127.0.0.1"),

//   };

//   let loopback = IpAddr{        //使用struct来存贮

//     kind:IpAddrKind::v6,

//     adress:String::from("::1"),

//   };

// }

//优点:不需要额外使用struct,每个变体都有不同类型以及相关数据源

// enum  IpAddr{

//   v4(u8,u8,u8,u8),

//   v6(String),

// }

//为枚举定义方法

//impl

enum Message{

  Quit,//无关联

  Move{x:i32,y:i32},//匿名结构体

  Write(String),

  ChangeColor(i32,i32,i32),

}

impl Message{

  fn call(&self){}

}

fn main(){

  let g = Message::Quit;

  let m = Message::Move{x: 12,y:34};

  let w = Message::Write(String::from("Hello"));

  let c = Message::ChangeColor(0,255,255);

  m.call();

}