// use std::io;//prelude

// use rand::Rng;

// use std::cmp::Ordering;

// const MAX\_POINTS:u32 = 100\_000;

// fn main() {

//     println!("猜数!");

//     let secret\_number = rand::thread\_rng().gen\_range(1,101);

//     println!("秘密数字是: {}",secret\_number);

//     loop{

//          println!("猜测一个数");

//     let mut guess = String::new();//rust中变量不可变,若要可变需加mut

//     io::stdin().read\_line(&mut guess)

//     .expect("无法读取行");

//     let guess:u32 = match guess.trim().parse(){

//         Ok(num) => num,

//         Err(\_) => continue,

//     };//隐藏shadow trim去掉空格和回车,parse将字符串转换为某种数字类型

//     println!("你猜测的数字:{}",guess);

//     match guess.cmp(&secret\_number){

//         Ordering::Less => println!("small!"),

//         Ordering::Greater => println!("big!"),

//         Ordering::Equal => {

//             println!("you win!");

//             break;

//         }

//     }

//     }

// }

// use std::io;

// fn main(){

//   let guess:u32 = "42".parse().expect("not a number");

//   println!("{}",guess);

// }

 //数据类型:u32,u64(无符号,范围是2^n - 1,如果溢出则会执行环绕操作,如u8范围是0-255,256变为零,257变为1,不会报错),

 //i32,i64,

 //f32,f64,(f64与f32速度差不多,但精度更高,默认f64)let y:f32 = 3.14;

 //加sum减difference乘product除quotient余reminder,支持浮点数取余

 //布尔类型:bool,true/false,

 //字符类型:char,单个Unicode字符,使用单引号,可以存比ASCII更大的字符,如'🤔',

 //元组tuple,不可变

 //元组中可以放任意数据类型,包括元组

//  fn main(){

//   let tup:(i32,f64,u8) = (500,6.4,1);

//   let (x,y,z) = tup;

//   println!("{},{},{}",x,y,z);

//   println!("{},{},{}",tup.0,tup.1,tup.2);

//  }

// 数组array,固定大小,元素类型必须一致

//使用栈存储数据或想保证固定数量的元素可以使用数组

//不如vector灵活

//如果索引超出数组范围,编译会通过,但运行时会panic,不允许访问相应地址的内存

//数组中要放置复合类型时,需要注意所有元素类型和长度一致

// fn main(){

//   let a = [1,2,3,4,5];

//   let b = [0;5];//数组的另一种初始化方式,用0填充数组,一共5个元素

//   let c : [i32; 5] = [1,2,3,4,5];

//   let first = a[0];

// }

//函数可以在任意地方声明

// fn main(){

//   println!("hello world");

//   another\_function(5);   argument实参

// }

// fn another\_function(x:i32){    parameter形参

//   println!("the value of x is {}",x);

// }

//函数可以有参数,分别为parameter,argument.在函数声明中必须声明每个参数类型

//函数由一系列语句组成,可选由一个表达式结束

//rust是一个基于表达式的语言

//语句是执行一些动作的指令

//表达式会计算产生 一个值

// fn main(){

//   let x = 5;//语句        整体也是个语句

// }

//函数定义也是语句

//语句不返回值,所以不可以使用let将1个语句赋给一个变量

//字面值是一个表达式,如6,5+6

//调用函数也是一个表达式,调用宏也是表达式

// fn main(){

//    let x = 5;

//    let y = {

//       let x = 1;      //创建了一个块,是一个表达式

//       x + 3     //块的返回值,或者说是块的值,结果输出4 ; 如果加上分号,则不会返回值,y是一个()空tuple,结果报错doesn't implement `std::fmt::Display`

//    };

//    println!("the value of y is {}",y);

// }

//函数返回值在->符号后面声明函数返回值类型,但不可以为函数返回值命名

// 在rust中,返回值就是函数最后一个表达式的值

// 若想提前返回,需要使用return关键字,并指定一个值

//  fn five(x:i32) -> i32{

//    x + 5

//  }

//  fn main(){

//    let x = five(5);

//    println!("the value of x is {}",x);

//  }

// if表达式

// 条件必须是bool类型,表达式的值是bool类型

// 与条件相关联的代码块叫分支arm

// fn main(){

//   let number = 7;

//   if number < 5{

//     println!("condition was true");

//   }

//   else{

//     println!("condition was false");

//   }

// }

//if是一个表达式,所以可以把它放在let语句等号右边

//if,else表达式每个能成为结果的分支返回类型必须一致

// fn main(){

//   let condition = true;

//   let number = if condition {5} else {6};//if true{5} else {6}

//   println!("the value of number is : {}",number);

// }

//loop循环

//loop关键字告诉rust反复执行一块代码直到叫停,使用break关键字

// fn main(){

//     let mut counter = 0;

//     let result = loop{

//       counter += 1;

//       if counter == 10{

//         break counter \* 2;

//       }

//     };

//     println!("the result is:{}",result);

// }

//while循环

// fn main(){

//   let mut number = 3;

//   while number != 0{

//     println!("{}!",number);

//     number = number - 1;

//   }

//   println!("LIFTOFF");

// }

//for循环

//遍历集合

//使用for循环更简洁紧凑,它可以针对集合中的么个元素来执行代码

//由于for循环的安全简洁,在rust使用最多

// fn main(){

//   let a = [10,20,30,40,50];

//   let mut index = 0;

//   // while index < 5{

//   //   println!("the value is :{}",a[index]);//while较慢

//   //   index = index + 1;

//   // }

//   for element in a.iter(){

//     println!("the value is :{}",element);//不会越界

//   }

// }

//range

//标准库提供

// 指定一个开始数字和一个结束数字,range可以生成他们之间的数字(不含结束),如(1..4)是1-3

// rev方法可以反转range

// fn main(){

//   for number in (1..4).rev(){

//     println!("{}!",number);

//   }

//   println!("LIFTOUT!");

// }

//所有权

//所有程序在运行时都必须管理它们使用计算机内存的方式

//所有权是rust最独特的特性,他让rust无需垃圾收集器就可以保障内存安全,

  //有些语言有垃圾收集机制,在程序运行时,它们会不断寻找不再使用的内存,如java,go

  //一些语言需要显式地分配和释放内存,如c,c++

// rust使用了第三种方式:

  //内存通过一个所有权系统管理,其中包含一组编译器在编译时检查的规则

  //在程序运行时,所有权不会减慢运行速度,以为rust把内存管理放到编译时,所以运行速度快

//stack(栈)和heap(堆)

//在rust这样系统级编程语言中,一个值是放在stack还是heap上对语言的行为以及这么做的原因有更大影响

//stack和heap都是可用的内存,但结构不同

//stack后进先出

//stack上的数据必须有已知且固定的大小编译时大小未知或可能发生改变的数据要放在heap上

//heap内存组织性较差

//把数据放入heap需要请求空间,操作系统在heap找到一块足够大的空间,把他们标记为在用,并返回一个指针,也就是这个空间的地址,即分配

//压入栈不需要分配,是连续放的,因为指针是固定大小的,所以可以把指针放在stack上,但需要用指针定位

//数据压到stack上比分配到heap要快,因为操作系统不需要寻找用来储存新数据的空间,它永远在stack顶端

//访问heap要比stack慢,因为需要通过指针来定位heap中的数据,所以访问速度慢

  //由于缓存的缘故,指令在内存中跳转次数越少则速度越快

  //而stack数据存放较近,所以访问速度快

//函数调用

//代码调用函数时,值被传入函数(包括指向heap指针),函数本地变量被压到stack上.函数结束后这些值会从stack上弹出

//所有权可以跟踪代码哪些部分正在使用heap的哪些数据

//可以最小化heap的重复数据量

//清理数据上未使用的数据避免空间不足

//所有权规则:

//每个值都有一个变量,这个变量是该值的所有者

//所有者超出作用域时这个值会被删除

//作用域scope

//string类型

//string类型更复杂,第二种字符串类型,可以被修改

//字面值:程序员写入的值,不可变

//在heap上分配,能够储存在编译时位置数量的文本

//使用from函数从字符串字面值创造出string类型

//let s = String::from("hello");

// fn main(){

//   let mut s = String::from("hello");

//   s.push\_str(",world!");//添加

//   println!("{}",s);

// }

// 内存和分配

// 字符串字面值在编译时就知道了,文本内容直接被硬编码在可执行文件中了.所以速度快,高效也是因为其不可变性

// String类型为了支持其可变性,需要在heap上分配内存来保存编译时的未知文本内容

  //操作系统必须在运行时请求内存,这步通过String::from来实现

  //用完String后,需要将内存返还给操作系统,在有垃圾收集器(GC)的语言中,GC会跟踪并清理不使用的内存,否则就要我们自己识别并手动将其返回

  //而rust中,对于某个值,当它走出变量作用域时,内存会自动将其交还给操作系统

//drop函数

//变量与数据交互方式:移动(move)

//   let x = 5;

//   let y = x;

// 整数类型大小固定,所以它们被压入栈中

// let s1 = String::from("hello");

// let s2 = s1;

/\*

  一个指向存放字符串内容的指针  ptr

  一个长度                    len

  一个容量                    capacity

  string由三部分组成,它们在stack上,存放字符串内容的部分在heap上

\*/

//由于s1赋给s2时,在栈上把指针,长度,容量都复制了一份,并没有复制heap上的数据,所以当变量离开作用域,rust会调用drop释放变量使用的heap内存,

//当s1s2都离开时会释放相同内存,就会引起二次释放的bug

//但rust会将s1失效,所有权移交,称为移动move

//若对heap上的数据想进行深度拷贝,可以使用clone方法

//let s2 = s2.clone();

//针对stack上的数据,使用复制copy

// fn main(){

//    let x = 5;

//    let y = x;

//    println!("x:{},y:{}",x,y);

// }

//因为stack上的数据在编译时已经确定了自己的大小,能将自己的数据完整存储在stack中,所以可以很快复制

//copy trait可以用于像整数这样的完全存放在stack上的类型,可以让变量在赋值后仍然可用(trait 接口)

//如果一个类型一部分实现了drop,则不被允许使用copy

//任何简单标量类型都是copy的,如整数,浮点数,布尔值,字符,元组(如果元素都是copy的),数组,指针

//需要分配内存或某种资源都不是copy的

//所有权与函数

//语义上,将值传给函数和赋给变量是类似的,(移动或复制)

// fn main(){

//   let s = String::from("Hello World");

//   take\_ownership(s);//s值被移动到函数里,之后不再有效

//   let x = 5;

//   makes\_copy(x);//x被移动到函数里,但因为是i32,实现copy,仍然有效

//   println!("x:{}",x);

// }

// fn take\_ownership(some\_string:String){

//   println!("{}",some\_string)

// }

// fn makes\_copy(some\_number:i32){

//   println!("{}",some\_number);

// }

//返回值与作用域,函数在返回值过程中同样也会发生所有权的转移

// fn main(){

//   let s1 = gives\_ownership();

//   let s2 = String::from("hello");

//   let s3 = takes\_and\_gives\_back(s2);

// }

// fn gives\_ownership() -> String{

//   let some\_string = String::from("hello");

//   some\_string

// }

// fn takes\_and\_gives\_back(a\_string:String) -> String{

//   a\_string//取得所有权,返回值时转移所有权

// }

//一个变量所有权总是会遵循同样模式:

  //一个值赋给其他变量时会发生移动

  //当一个包含heap数据的变量离开作用域时,它的值会被drop函数清理,除非所有权转移

//若要让函数获得某个值,但是不获得所有权

// fn main(){

//   let s1 = String::from("hello");

//   let(s1,len) = calculate\_length(s1);

//   println!("the length of '{}' is {}.",s1,len);//相当于把所有权交出去在还回来

// }

// fn calculate\_length(s:String) -> (String,usize){//usize是和架构有关的无符号整数类型

//   let length = s.len();

//   (s,length)

// }

//引用和借用

// fn main(){

//   let s1 = String::from("hello");

//   let len = calucate\_length(&s1);

//   println!("the length of '{}' is {}",s1,len);//不被清理掉,没有拿走所有权

// }

// fn calucate\_length(s:&String) -> usize{

//   s.len()

// }

//&表示引用:允许你引用值而不获得所有权,所以不可以接push等操作

//通过引用作为函数参数的行为叫借用

//和变量一样不可变,可以用mut  (&mut),就可以push了

//在特定作用域对一块数据只能有一个可变引用,好处是在编译时防止数据竞争

//(数据竞争:两个或多个指针访问同一个数据,至少有一个指针用于写入数据,没有任何机制来同步对数据据的访问)

//不能同时有一个可变引用和不可变引用,因为会改值

//悬空引用

//即野指针,rust会报错

//切片slice

//rust的另一种不掌握所有权的数据类型

// fn main(){

//   let mut s = String::from("hello world");

//   let worldIndex = first\_word(&s);

//   println!("{}",wordIndex);

// }

// fn first\_word(s:&String) -> usize{

//   let bytes = s.as\_bytes();

//   for(i,&item) in bytes.iter().enumerate(){ //iter方法创建了一个迭代器,依次返回集合中的每个元素 .

//                                             //enumerate 枚举 ,把iter结果进行包装,作为元组一部分进行返回,第一个变量是索引,第二个变量是值也就是字节

//                                             //使用模式匹配对,分别生命两个变量对元组进行结构

//       if item == b' '{

//         return i;

//       }

//   }

//   s.lens()

// }

//但是会出错,这个函数是吧索引位置返回给了调用者,索引位置独立于字符串,函数返回后无法保证有效性,如在println!("{}",wordIndex);前s.clear();

//字符串切片 解决这个问题  (&str)

//字符串切片指字符串中一部分内容的引用

// fn main(){

//   let s = String::from("hello world");

//   let hello = &s[0..5];//第一个可以不写

//   let world = &s[6..11];//如果是最后一个可以换成s.len(),或者不写

//   let whole = &s[..];

// }

//字符串切片范围索引必须发生在有效的utf-8字符边界内

//如果尝试从多字节的字符串中创造切片则程序会报错或者推退出

// fn main(){

//   let mut s = String::from("hello world");

//   let worldIndex = first\_word(&s);//发生了不可变引用,所以已经不能对s进行修改了

//   println!("{}",wordIndex);

// }

// fn first\_word(s:&String) -> &str{

//   let bytes = s.as\_bytes();

//   for(i,&item) in bytes.iter().enumerate(){ //iter方法创建了一个迭代器,依次返回集合中的每个元素 .

//                                             //enumerate 枚举 ,把iter结果进行包装,作为元组一部分进行返回,第一个变量是索引,第二个变量是值也就是字节

//                                             //使用模式匹配对,分别生命两个变量对元组进行结构

//       if item == b' '{

//         return &s[..i];

//       }

//   }

//   &s[..]

// }

//字符串字面值就是一个切片

//直接存储在了二进制程序中

//把字符串切片作为参数进行传递

//把fn first\_word(s:&String) -> &str换成fn first\_word(s:&str) -> &str

//使用字符串切片直接调用该函数,使用String可以穿造一个完整string切片来调用函数

//会让API更通用,不会造成功能损失

//其他类型切片

fn main(){

  let a = [1,2,3,4,5];

  let slice = &a[1..3];//slice为&[132],还能存储长度

}