**جواب سوالات پروژه درس برنامه نویسی سیستمی**

**دانشجو ترم 6**

**کارشناسی ناپیوسته**

**مهندسی حرفه ای کامپیوتر نرم افزار**

**یاسین شهریاری کوتک**

**1سوال یک:::-**زبان برنامه‌نویسی **Rust** یک زبان برنامه‌نویسی سیستم است که برای **سرعت، ایمنی و همزمانی** طراحی شده است. این زبان در ابتدا توسط **Mozilla** در سال 2010 معرفی شد و هدف آن ارائه ابزاری برای نوشتن برنامه‌های با کارایی بالا، ایمن و بدون خطاهای رایج در برنامه‌نویسی سیستم است.

**ویژگی‌های اصلی زبان Rust عبارتند از:**

1. **ایمنی حافظه بدون جمع‌آوری زباله (Garbage Collection):**
   * یکی از ویژگی‌های کلیدی Rust سیستم مدیریت حافظه است که از **مالکیت** و **امانت‌داری** (Borrowing) برای جلوگیری از خطاهای حافظه مثل دسترسی به اشاره‌گرهای معیوب، overflows و استفاده از حافظه آزاد شده استفاده می‌کند. این سیستم از جمع‌آوری زباله (garbage collection) استفاده نمی‌کند، اما با این حال ایمنی حافظه را به شکل موثری تأمین می‌کند.
2. **همزمانی بدون مشکلات داده‌ای (Data Race):**
   * Rust طراحی شده است تا **برنامه‌های همزمان** (concurrent) را به‌طور ایمن و بدون مشکلات رایج همزمانی مانند **داده‌های مشترک** (data races) مدیریت کند. این ویژگی از زمان کامپایل، مشکلات مربوط به دسترسی همزمان به داده‌ها را شناسایی کرده و رفع می‌کند.
3. **عملکرد بالا:**
   * Rust به‌طور مشابه به زبان‌های **C و C++** بهینه است و کد به‌صورت مستقیم به کد ماشین ترجمه می‌شود. این ویژگی امکان نوشتن برنامه‌هایی با عملکرد بسیار بالا را فراهم می‌آورد.
4. **ابستراکت‌های بدون هزینه:**
   * Rust ابستراکت‌هایی مانند توابع مرتبه بالا و الگوهای تطابق (pattern matching) را فراهم می‌آورد که بدون هزینه اجرایی اضافی استفاده می‌شوند. این ویژگی به برنامه‌نویس اجازه می‌دهد تا از امکانات پیشرفته زبان استفاده کرده و همچنان عملکرد بالای کد را حفظ کند.
5. **پشتیبانی از برنامه‌نویسی تابعی و دستوری:**
   * Rust از هر دو سبک **برنامه‌نویسی تابعی** و **دستوری** پشتیبانی می‌کند و این انعطاف‌پذیری به برنامه‌نویسان این امکان را می‌دهد که بهترین روش‌ها را بسته به نیاز خود انتخاب کنند.
6. **ابزارها و اکوسیستم قوی:**
   * Rust ابزارهای بسیاری مانند cargo برای مدیریت پروژه‌ها، rustfmt برای فرمت‌بندی کد و rustc برای کامپایل کردن را فراهم می‌کند. این ابزارها به توسعه‌دهندگان کمک می‌کنند تا به راحتی برنامه‌های Rust را نوشته، آزمایش و مدیریت کنند.
7. **پلتفرم‌های متعدد:**
   * Rust می‌تواند بر روی بسیاری از پلتفرم‌ها اجرا شود، از جمله **ویندوز، لینوکس، macOS** و حتی **WebAssembly** برای توسعه وب.

### مزایا و کاربردها:

Rust به دلیل ایمنی بالا و عملکرد بهینه، در زمینه‌های مختلفی از جمله **برنامه‌نویسی سیستم‌ها، توسعه بازی، توسعه وب با WebAssembly، و بلاک‌چین** مورد استفاده قرار می‌گیرد. زبان Rust به خصوص برای پروژه‌هایی که نیاز به کنترل دقیق روی منابع سیستم دارند، مانند سیستم‌عامل‌ها و برنامه‌های کاربردی با کارایی بالا، مناسب است.

### نتیجه‌گیری:

Rust یک زبان برنامه‌نویسی قدرتمند و مدرن است که در عین حفظ سرعت و کارایی بالا، به توسعه‌دهندگان امکان می‌دهد تا کدهایی ایمن و بدون خطاهای حافظه بنویسند.

**2سوال دو :::** ویژگی‌های زبان برنامه‌نویسی **Rust** به اختصار عبارتند از:

1. **ایمنی حافظه:**
   * Rust از سیستم **مالکیت** و **امانت‌داری** برای جلوگیری از خطاهای رایج حافظه مانند استفاده از اشاره‌گرهای معیوب و آزاد شدن حافظه استفاده شده (use-after-free) بهره می‌برد، بدون نیاز به **جمع‌آوری زباله** (garbage collection).
2. **عملکرد بالا:**
   * Rust به مانند زبان‌های C و C++ بهینه است و کد آن مستقیماً به کد ماشین تبدیل می‌شود، به این معنی که عملکرد بالایی دارد.
3. **همزمانی ایمن:**
   * Rust امکان نوشتن برنامه‌های همزمان (concurrent) را بدون نگرانی از مشکلات داده‌ای (data races) فراهم می‌آورد.
4. **ابستراکت‌های بدون هزینه:**
   * زبان Rust از ابستراکت‌هایی مانند توابع مرتبه بالا و الگوهای تطابق (pattern matching) استفاده می‌کند که بدون هزینه اضافی در زمان اجرا به کار می‌روند.
5. **پشتیبانی از برنامه‌نویسی تابعی و دستوری:**
   * Rust از برنامه‌نویسی تابعی و دستوری پشتیبانی می‌کند و این انعطاف‌پذیری را به برنامه‌نویسان می‌دهد تا به بهترین روش ممکن کدنویسی کنند.
6. **ابزارها و اکوسیستم غنی:**
   * Rust شامل ابزارهایی مانند cargo برای مدیریت پروژه‌ها، rustfmt برای فرمت‌بندی کد، و rustc برای کامپایل کردن کد است که به توسعه‌دهندگان کمک می‌کند پروژه‌های خود را به راحتی مدیریت کنند.
7. **کراس‌پلتفرم:**
   * Rust می‌تواند بر روی سیستم‌عامل‌های مختلف از جمله **ویندوز، لینوکس، macOS** و **WebAssembly** اجرا شود.
8. **جامعه و منابع فعال:**
   * Rust دارای یک جامعه فعال و منابع آموزشی و کتابخانه‌های وسیع است که به توسعه‌دهندگان کمک می‌کند در توسعه نرم‌افزارهای خود سریع‌تر پیشرفت کنند.
9. **ایمنی حافظه (Memory Safety):**
   * Rust با استفاده از سیستم **مالکیت** (ownership) و **امانت‌داری** (borrowing)، از بروز خطاهایی مانند **دسترسی به حافظه آزاد شده** (use-after-free) و **دسترسی به حافظه نامعتبر** (null pointers) جلوگیری می‌کند. این کار بدون نیاز به **جمع‌آوری زباله** (garbage collection) انجام می‌شود و باعث افزایش ایمنی کد می‌شود.
10. **عملکرد بالا:**
    * Rust مانند زبان‌های **C و C++**، کد را مستقیماً به **کد ماشین** ترجمه می‌کند، بنابراین عملکرد بسیار بالایی دارد. این ویژگی آن را برای نوشتن برنامه‌های با کارایی بالا مانند سیستم‌عامل‌ها و بازی‌ها مناسب می‌سازد.
11. **همزمانی ایمن (Concurrency):**
    * Rust به طور خاص برای پشتیبانی از **همزمانی** طراحی شده است. این زبان از **مدیریت ایمن داده‌ها** در محیط‌های چندپردازشی (multi-threaded) پشتیبانی می‌کند، به گونه‌ای که از **داده‌های مشترک** و **داده‌های همزمان** (data races) جلوگیری می‌کند.
12. **ابستراکت‌های بدون هزینه (Zero-Cost Abstractions):**
    * Rust امکان استفاده از ویژگی‌هایی مانند **توابع مرتبه بالا** و **الگوهای تطابق** (pattern matching) را بدون اینکه تأثیری بر عملکرد برنامه داشته باشد، فراهم می‌آورد. این ویژگی‌ها به توسعه‌دهندگان اجازه می‌دهند تا کدی ساده و قابل فهم بنویسند.
13. **پشتیبانی از برنامه‌نویسی تابعی و دستوری:**
    * Rust ترکیبی از **برنامه‌نویسی دستوری** (imperative) و **تابعی** (functional) است. این زبان از **توابع مرتبه بالا**، **الگوهای تطابق** و **immutable data** پشتیبانی می‌کند که سبک‌های مختلف برنامه‌نویسی را پوشش می‌دهد.
14. **مدیریت پکیج‌ها و وابستگی‌ها (Cargo):**
    * Rust از سیستم **Cargo** برای مدیریت پروژه‌ها و وابستگی‌ها استفاده می‌کند. این ابزار به طور خودکار بسته‌ها (crates) را دانلود، نصب و مدیریت می‌کند.
15. **کامپایلر قوی:**
    * کامپایلر Rust به نام **rustc** بسیار قدرتمند است و بسیاری از مشکلات رایج کدنویسی را در زمان کامپایل شناسایی و گزارش می‌دهد. این به کاهش خطاها در زمان اجرای برنامه کمک می‌کند.
16. **کتابخانه‌ها و اکوسیستم غنی (Crates.io):**
    * زبان Rust دارای یک **اکوسیستم وسیع** از کتابخانه‌ها (crates) است که می‌توانند برای پروژه‌های مختلف استفاده شوند. **Crates.io** به عنوان مخزن مرکزی برای به اشتراک‌گذاری و استفاده از این کتابخانه‌ها عمل می‌کند.
17. **کراس‌پلتفرم:**
    * Rust می‌تواند بر روی سیستم‌عامل‌های مختلف مانند **ویندوز، لینوکس، macOS** و حتی **WebAssembly** اجرا شود. این ویژگی آن را برای توسعه برنامه‌های قابل حمل مناسب می‌سازد.
18. **جامعه فعال:**
    * Rust دارای یک **جامعه فعال** و **منابع آموزشی** متنوع است که به توسعه‌دهندگان کمک می‌کند تا مشکلات خود را حل کنند و با جدیدترین روندهای این زبان آشنا شوند.
19. **مدیریت خطاها (Error Handling):**
    * Rust از **نتایج** (Result) و **اختیاری‌ها** (Option) برای مدیریت خطاها استفاده می‌کند که اجازه می‌دهد خطاها به‌صورت ایمن و قابل کنترل مدیریت شوند.

این ویژگی‌ها باعث شده‌اند که Rust به یکی از محبوب‌ترین زبان‌ها برای برنامه‌نویسی سیستم‌ها، برنامه‌های همزمان، و توسعه نرم‌افزارهای با کارایی بالا تبدیل شود.

**سوال سه::3**

زبان Rust در چندین زمینه مختلف کاربرد زیادی دارد که به دلیل ویژگی‌های خاص خود، مانند سرعت بالا، امنیت حافظه و پشتیبانی از چندپردازشی (Concurrency) به آن اهمیت ویژه‌ای بخشیده است. در اینجا به برخی از زمینه‌های اصلی کاربرد Rust اشاره می‌کنیم:

**1. سیستم‌های نرم‌افزاری و نرم‌افزارهای سیستمی**

* Rust به دلیل سرعت بالای اجرا و کنترل دقیق بر منابع سیستم، در زمینه توسعه نرم‌افزارهای سیستمی و برنامه‌های نزدیک به سیستم (System-level programming) مانند **کرنل‌ها، درایورها، سیستم‌عامل‌ها، و نرم‌افزارهای شبکه** کاربرد دارد.
* این زبان به دلیل مدیریت ایمن حافظه بدون نیاز به جمع‌آوری زباله (Garbage Collection) بسیار مناسب برای این نوع برنامه‌ها است.

**2. توسعه بازی‌ها**

* Rust برای توسعه بازی‌ها و موتورهای بازی‌سازی نیز کاربرد دارد. برخی از شرکت‌ها از Rust برای توسعه بخش‌های خاصی از بازی‌های خود استفاده می‌کنند، زیرا این زبان می‌تواند عملکرد بالا را در کنار امنیت حافظه فراهم کند.
* **Amethyst** یکی از موتورهای بازی‌سازی نوشته‌شده با Rust است.

**3. برنامه‌های چندپردازشی و شبکه‌ای**

* Rust ویژگی‌هایی مانند **ساختارهای داده ایمن در چندپردازشی** و مدیریت موثر منابع را به خوبی پشتیبانی می‌کند که آن را برای توسعه **برنامه‌های شبکه‌ای** و **توزیع‌شده** و همچنین **برنامه‌های با نیازهای پردازش موازی** مناسب می‌سازد.

**4. توسعه وب (WebAssembly)**

* Rust به طور گسترده در **توسعه برنامه‌های WebAssembly (Wasm)** مورد استفاده قرار می‌گیرد. با استفاده از WebAssembly، برنامه‌های Rust می‌توانند در مرورگرها اجرا شوند و عملکرد بالاتری نسبت به جاوااسکریپت داشته باشند.
* پروژه‌هایی مانند **Yew** و **Seed** از Rust برای ساخت اپلیکیشن‌های وب استفاده می‌کنند.

**5. تحلیل داده‌ها و پردازش‌های سنگین**

* به دلیل سرعت بالای Rust، از آن در پروژه‌های **تحلیل داده‌ها**، **پردازش‌های علمی** و **ماشین لرنینگ** استفاده می‌شود.
* البته در این زمینه‌ها، Rust معمولاً به همراه زبان‌هایی مانند Python یا Julia استفاده می‌شود تا بتوان از کتابخانه‌های تخصصی آن‌ها بهره برد.

**6. ابزارهای DevOps و اسکریپت‌نویسی**

* Rust در زمینه توسعه **ابزارهای DevOps** مانند مدیریت پیکربندی و اتوماسیون سیستم‌ها نیز استفاده می‌شود. برنامه‌هایی مانند **Cargo** (مدیر بسته‌های Rust) و **Rustup** (ابزار نصب و مدیریت نسخه‌های Rust) نمونه‌هایی از ابزارهای محبوب Rust در این زمینه هستند.

**7. امنیت و رمزنگاری**

* Rust به دلیل ویژگی‌های ایمنی حافظه، به طور خاص در زمینه‌های مرتبط با **امنیت سایبری** و **رمزنگاری** مورد توجه قرار گرفته است.
* بسیاری از پروژه‌های مربوط به پیاده‌سازی الگوریتم‌های رمزنگاری و ابزارهای امنیتی، از Rust به عنوان زبان انتخابی استفاده می‌کنند.

**8. پروژه‌های Open Source**

* Rust به دلیل پشتیبانی از جامعه فعال و قابلیت همکاری با دیگر زبان‌ها در پروژه‌های **متن‌باز** بسیار محبوب است.
* پروژه‌هایی مانند **Servo** (موتور رندر وب) و **RustOS** (سیستم‌عامل مبتنی بر Rust) نمونه‌های مشهوری از پروژه‌های متن‌باز نوشته‌شده با Rust هستند.

در مجموع، زبان Rust به دلیل ترکیب ویژگی‌هایی مانند **عملکرد بالا، ایمنی حافظه، و توانایی‌های چندپردازشی** در بسیاری از زمینه‌های فنی و پیچیده کاربرد دارد و محبوبیت آن روز به روز در حال افزایش است.

**سوال4::**

زبان‌های **Rust**، **C** و **C++** همگی زبان‌های سطح پایین هستند که برای توسعه نرم‌افزارهایی که نیاز به عملکرد بالا دارند، مورد استفاده قرار می‌گیرند. این زبان‌ها شباهت‌های زیادی دارند، اما در عین حال تفاوت‌های قابل توجهی نیز میان آن‌ها وجود دارد. در اینجا به بررسی تفاوت‌های اصلی این سه زبان پرداخته می‌شود:

**1. ایمنی حافظه (Memory Safety)**

* **Rust:** یکی از ویژگی‌های برجسته Rust این است که به طور پیش‌فرض ایمنی حافظه را تضمین می‌کند. این زبان از **سیستم مالکیت** (Ownership System) برای مدیریت حافظه استفاده می‌کند. این سیستم باعث می‌شود که بسیاری از مشکلات رایج مربوط به حافظه، مانند دسترسی به حافظه آزاد شده یا نشت حافظه، به طور خودکار در زمان کامپایل شناسایی و رفع شوند.
* **C/C++:** در C و C++، مدیریت حافظه به طور دستی انجام می‌شود. این بدین معنی است که برنامه‌نویس باید خود حافظه را تخصیص دهد و آزاد کند، که این امر می‌تواند به راحتی منجر به بروز مشکلاتی مانند **نشت حافظه** (Memory Leaks)، **دسترسی به حافظه آزاد شده** (Dangling Pointers) یا **خطاهای رقابتی** (Race Conditions) شود.

**2. مدیریت حافظه و جمع‌آوری زباله (Garbage Collection)**

* **Rust:** Rust هیچ‌گونه جمع‌آوری زباله ندارد. در عوض، از سیستم مالکیت برای مدیریت حافظه استفاده می‌کند که به طور خودکار حافظه غیر ضروری را پس از اتمام استفاده، آزاد می‌کند. این امر باعث می‌شود که برنامه‌های Rust سریع و کارآمد باشند.
* **C/C++:** در C و C++، مدیریت حافظه کاملاً بر عهده برنامه‌نویس است. اگرچه C++ به شما اجازه می‌دهد از مکانیزم‌هایی مانند destructors برای مدیریت حافظه استفاده کنید، اما همچنان مسئولیت‌ها به عهده خود برنامه‌نویس باقی می‌ماند.

**3. عملکرد و سرعت**

* **Rust:** Rust به دلیل اینکه به کامپایلر خود اجازه می‌دهد بهینه‌سازی‌های زیادی روی کد انجام دهد، به طور کلی عملکرد بسیار بالایی دارد. سیستم مالکیت Rust ممکن است در برخی شرایط منجر به پیچیدگی‌های اضافی شود، اما همچنان در مقایسه با زبان‌هایی مثل C و C++ عملکردی مشابه یا حتی بهتر دارد.
* **C/C++:** C و C++ به طور تاریخی زبان‌هایی هستند که برای دستیابی به حداکثر سرعت و کارایی طراحی شده‌اند. این زبان‌ها کنترل دقیقی بر روی منابع سخت‌افزاری دارند، که این باعث می‌شود سرعت آنها بسیار بالا باشد. C++ به خصوص قابلیت‌های گسترده‌ای برای مدیریت حافظه و کنترل منابع فراهم می‌آورد.

**4. سینتکس (Syntax)**

* **Rust:** Rust از یک سینتکس مدرن‌تر و ساده‌تر نسبت به C و C++ بهره می‌برد. بسیاری از مشکلات معمول در زبان‌های C/C++ از طریق طراحی دقیق‌تر Rust حل شده‌اند. همچنین Rust ویژگی‌هایی مانند **الگوهای match** و **توابع تو در تو** (closures) را به شکلی ساده‌تر از C++ ارائه می‌دهد.
* **C/C++:** سینتکس C و C++ بیشتر به سبک قدیمی‌تر زبان‌های برنامه‌نویسی است و نیاز به توجه بیشتری به جزئیات دارد. در C++ این زبان پیچیده‌تر است، زیرا ویژگی‌هایی مانند **شیء‌گرایی (OOP)**، **الگوها (Templates)** و **خصوصیات پیشرفته حافظه** نیازمند کدهای پیچیده‌تر و آگاهی از اصول پیچیده‌تر هستند.

**5. مدیریت هم‌زمانی (Concurrency)**

* **Rust:** Rust به طور طبیعی از هم‌زمانی ایمن پشتیبانی می‌کند. این زبان ویژگی‌هایی مانند **مستقل بودن داده‌ها** (Data Races) و **اعمال قفل‌ها** (Locks) را به شکلی خودکار و بدون نیاز به دستورات اضافی مدیریت می‌کند.
* **C/C++:** در C و C++ هم‌زمانی به طور کامل به عهده برنامه‌نویس است. کتابخانه‌هایی مانند **pthread** در C و **std::thread** در C++ برای مدیریت هم‌زمانی ارائه شده‌اند، اما این فرآیند می‌تواند پیچیده و مستعد خطا باشد، به خصوص زمانی که داده‌ها در بین چندین نخ به اشتراک گذاشته شوند.

**6. ابزارهای پشتیبانی و توسعه**

* **Rust:** Rust از نظر ابزارهای توسعه به خوبی پشتیبانی می‌شود. ابزارهایی مانند **Cargo** (مدیر بسته‌ها و بیلد سیستم)، **Rustfmt** (برای قالب‌بندی کد) و **Clippy** (برای تحلیل کد) به طور پیش‌فرض در دسترس هستند. این ابزارها تجربه توسعه را در Rust راحت‌تر و سریع‌تر می‌کنند.
* **C/C++:** ابزارهای C و C++ نیز به خوبی توسعه یافته‌اند، اما ممکن است نیاز به تنظیمات بیشتری داشته باشند. همچنین در C++، مسائل مربوط به مدیریت وابستگی‌ها و بیلد سیستم‌ها پیچیده‌تر است.

**7. کتابخانه‌ها و اکوسیستم**

* **Rust:** Rust به دلیل نسبتاً جدید بودن آن، اکوسیستم بزرگی ندارد، اما به سرعت در حال رشد است. بسته‌ها و کتابخانه‌های زیادی از طریق **Crates.io** (مخزن بسته‌های Rust) در دسترس هستند.
* **C/C++:** C و C++ دارای اکوسیستم‌های بسیار بزرگی هستند که شامل هزاران کتابخانه و ابزار برای انواع مختلف کاربردها می‌شود. این زبان‌ها به مدت طولانی‌تری در صنعت استفاده شده‌اند، بنابراین منابع گسترده‌ای دارند.

**8. توسعه و یادگیری**

* **Rust:** به دلیل پیچیدگی‌های سیستم مالکیت و نیاز به دقت در مدیریت منابع، Rust ممکن است برای افراد تازه‌کار چالش‌برانگیز باشد. با این حال، مستندات و منابع آموزشی بسیار خوبی برای یادگیری Rust وجود دارد.
* **C/C++:** C و C++ به طور سنتی زبان‌های چالش‌برانگیز برای یادگیری هستند، به خصوص به دلیل پیچیدگی‌های مربوط به مدیریت حافظه و ویژگی‌های پیشرفته مانند پویایی حافظه و الگوهای برنامه‌نویسی شی‌گرا.

**نتیجه‌گیری:**

* **Rust** به دلیل ویژگی‌های ایمنی حافظه، مدیریت بهتر هم‌زمانی و ابزارهای پشتیبانی پیشرفته به یک گزینه جذاب برای توسعه نرم‌افزارهای سیستم‌محور و عملکرد بالا تبدیل شده است.
* **C** و **C++** همچنان در بسیاری از زمینه‌ها، به خصوص در مواردی که نیاز به کنترل دقیق بر منابع و عملکرد بالا وجود دارد، کاربرد دارند. اما از نظر ایمنی و سادگی کدنویسی، Rust مزیت‌های قابل توجهی ارائه می‌دهد.

**سوال5::**

زبان **Rust** به دلیل طراحی مدرن و تمرکز بر ایمنی حافظه و کارایی، به یکی از محبوب‌ترین زبان‌های برنامه‌نویسی تبدیل شده است. با این حال، مانند هر زبان دیگری، محدودیت‌ها و چالش‌هایی دارد. در ادامه به برخی از این محدودیت‌ها اشاره می‌شود:

### 1. ****منحنی یادگیری شیب‌دار****

یکی از بزرگ‌ترین چالش‌های Rust، **پیچیدگی یادگیری** آن است. مدیریت قوانین مالکیت (ownership)، وام‌گیری (borrowing) و طول عمر (lifetimes) برای برنامه‌نویسانی که تازه با این زبان کار می‌کنند، دشوار است.

### 2. ****زمان کامپایل طولانی****

به دلیل ویژگی‌های ایمنی و بهینه‌سازی‌های پیشرفته، زمان کامپایل در Rust معمولاً نسبت به زبان‌هایی مثل C یا C++ بیشتر است. این مسئله می‌تواند بر چرخه توسعه و آزمایش تأثیر منفی بگذارد.

### 3. ****کمبود کتابخانه‌ها و ابزارهای بالغ****

اگرچه اکوسیستم Rust به سرعت در حال رشد است، برخی از **کتابخانه‌ها و ابزارها** به اندازه‌ی زبان‌های قدیمی‌تری مثل Python یا Java بالغ نیستند. برای پروژه‌های خاص، ممکن است یافتن کتابخانه‌های مناسب چالش‌برانگیز باشد.

### 4. ****محدودیت‌های کاربرد در برخی حوزه‌ها****

Rust برای توسعه سیستم‌های سطح پایین و برنامه‌های کارایی‌محور عالی است، اما برای **برخی کاربردهای خاص**، مانند برنامه‌نویسی وب یا یادگیری ماشین، هنوز نسبت به زبان‌های دیگر ابزارهای کمتری دارد. با وجود کتابخانه‌هایی مانند Rocket برای توسعه وب، یا Tch و ndarray برای یادگیری ماشین، Rust همچنان به اندازه برخی زبان‌ها محبوبیت و ابزار ندارد.

### 5. ****کاهش تولید کد پویا****

برخلاف زبان‌هایی مثل Python یا JavaScript که تولید **کدهای پویا** (dynamic code generation) را به راحتی فراهم می‌کنند، Rust به دلیل ویژگی‌های ایمنی خود چنین قابلیتی را محدودتر ارائه می‌دهد.

### 6. ****کمبود منابع انسانی متخصص****

از آنجا که Rust زبان جدیدتری است، تعداد برنامه‌نویسانی که به طور کامل به این زبان مسلط باشند نسبتاً کم‌تر است، و این می‌تواند برای شرکت‌ها در استخدام نیروهای متخصص چالش ایجاد کند.

### 7. ****تعامل با کدهای C و C++****

اگرچه Rust از **واسطه‌های خارجی (FFI)** برای تعامل با کدهای C و C++ پشتیبانی می‌کند، این فرآیند می‌تواند پیچیده‌تر از آنچه در زبان‌های دیگر است باشد و ایمنی حافظه در این موارد به طور کامل تضمین نمی‌شود.

### نتیجه‌گیری

Rust با وجود محدودیت‌ها و چالش‌های ذکر شده، یکی از **ایمن‌ترین و کارآمدترین زبان‌های برنامه‌نویسی مدرن** است. اگرچه یادگیری آن زمان‌بر است، مزایای آن در پروژه‌های بلندمدت می‌تواند بسیار ارزشمند باشد.

**سوال6::**

**Cargo** ابزار مدیریت بسته‌ها و سیستم ساخت (build system) در زبان برنامه‌نویسی **Rust** است که نقش حیاتی در توسعه پروژه‌ها دارد. این ابزار به طور پیش‌فرض همراه با نصب Rust ارائه می‌شود و کار توسعه‌دهندگان را برای مدیریت وابستگی‌ها و ساخت کد بسیار آسان می‌کند. در ادامه، قابلیت‌ها و ویژگی‌های مهم Cargo را توضیح می‌دهیم:

### 1. ****ایجاد پروژه جدید****

برای شروع یک پروژه جدید در Rust، می‌توانید از دستور زیر استفاده کنید:

bash

Copy code

cargo new project\_ name

این دستور یک پروژه با ساختار فایل مناسب، شامل پوشه‌های src و فایل Cargo Toml (برای مدیریت وابستگی‌ها) ایجاد می‌کند.

### 2. ****ساخت و کامپایل پروژه****

برای کامپایل پروژه، کافی است دستور زیر را اجرا کنید:

bash

Copy code

cargo build

این دستور کد منبع را کامپایل کرده و فایل‌های اجرایی تولید می‌کند.

برای ساخت نسخه بهینه‌شده (برای تولید) از دستور زیر استفاده می‌شود:

bash

Copy code

cargo build --release

### 3. ****اجرای پروژه****

برای اجرای پروژه پس از ساخت:

bash

Copy code

cargo run

این دستور کد را کامپایل کرده و برنامه را اجرا می‌کند.

### 4. ****مدیریت وابستگی‌ها****

فایل Cargo.toml برای مدیریت وابستگی‌های خارجی استفاده می‌شود. اضافه کردن وابستگی جدید به این فایل به شکل زیر است:

toml

Copy code

[dependencies]

serde = "1.0"

سپس با اجرای دستور زیر وابستگی‌ها نصب می‌شوند:

bash

Copy code

cargo build

### 5. ****تست کردن کد****

Cargo امکان اجرای تست‌ها را به شکل ساده فراهم می‌کند:

bash

Copy code

cargo test

این دستور تمامی تست‌های موجود در پروژه را اجرا می‌کند.

### 6. ****فرمت کردن کد****

برای فرمت کردن کد بر اساس استانداردهای Rust از دستور زیر استفاده می‌شود:

bash

Copy code

cargo fmt

### 7. ****چک کردن کد بدون ساخت****

برای بررسی سریع کد و یافتن خطاهای احتمالی بدون کامپایل کامل از دستور زیر استفاده می‌شود:

bash

Copy code

cargo check

این دستور نسبت به cargo build بسیار سریع‌تر عمل می‌کند.

### 8. ****مستندسازی****

برای تولید مستندات خودکار از کدهای Rust استفاده می‌شود:

bash

Copy code

cargo doc --open

این دستور مستندات پروژه را ایجاد کرده و به طور خودکار در مرورگر باز می‌کند.

### نتیجه‌گیری

Cargo یکی از قوی‌ترین ابزارهای مدیریت پروژه و وابستگی‌ها در دنیای زبان‌های برنامه‌نویسی است که به دلیل یکپارچگی عمیق با Rust، توسعه و مدیریت پروژه‌ها را بسیار آسان و کارآمد می‌کند.

**س7::**

فایل  **Cargo lock**  یکی از فایل‌های کلیدی در پروژه‌های Rust است که توسط ابزار **Cargo** ایجاد و مدیریت می‌شود. این فایل وظیفه ثبت دقیق نسخه‌های وابستگی‌های پروژه را بر عهده دارد. در ادامه به اهمیت، عملکرد و دلایل استفاده از این فایل می‌پردازیم.

### ****وظیفه اصلی فایل Cargo .lock****

وظیفه اصلی این فایل **قفل کردن نسخه‌های دقیق وابستگی‌های پروژه** است تا در هر بار کامپایل یا بین سیستم‌های مختلف، از همان نسخه‌های یکسان کتابخانه‌ها استفاده شود. این کار تضمین می‌کند که کد پروژه به صورت قابل پیش‌بینی اجرا شود و مشکلات ناشی از ناسازگاری نسخه‌ها به حداقل برسد.

### ****مقایسه با فایل Cargo toml****

| **Cargo Toml** | **Cargo. Lock** |
| --- | --- |
| وابستگی‌های پروژه را با نسخه‌های اصلی ( semver) تعریف می‌کند. | نسخه‌های دقیق و دقیقاً مشخص از هر وابستگی را ذخیره می‌کند. |
| توسط توسعه‌دهنده به‌صورت دستی تغییر می‌کند. | به‌طور خودکار توسط Cargo ایجاد و مدیریت می‌شود. |
| ممکن است فقط نسخه‌های اصلی (مانند serde = "1.0") را مشخص کند. | نسخه کامل (مانند serde = "1.0.130") را ذخیره می‌کند. |

### ****چرا فایل Cargo. lock مهم است؟****

1. **تضمین تکرارپذیری ساخت**  
   با استفاده از این فایل، پروژه شما همواره از نسخه‌های دقیق مشخصی از وابستگی‌ها استفاده می‌کند، حتی اگر نسخه‌های جدیدی از آن‌ها منتشر شده باشند.
2. **پایداری بین سیستم‌ها**  
   هنگامی که پروژه روی سیستم‌های مختلف اجرا می‌شود، وجود Cargo. lock باعث می‌شود وابستگی‌ها یکسان باقی بمانند و رفتار برنامه تغییر نکند.
3. **مدیریت به‌روزرسانی‌ها**  
   به‌روزرسانی وابستگی‌ها به‌طور دقیق قابل کنترل است. با اجرای cargo update، نسخه‌های جدیدتر به فایل Cargo. lock وارد می‌شوند، اما این تغییر فقط با تصمیم صریح توسعه‌دهنده اتفاق می‌افتد.

### ****آیا باید Cargo. lock در کنترل نسخه ( git ) قرار گیرد؟****

* **برای برنامه‌ها**: باید فایل Cargo. lock را در سیستم کنترل نسخه ذخیره کنید، زیرا این فایل تضمین می‌کند که همه اعضای تیم توسعه از نسخه‌های یکسان وابستگی‌ها استفاده می‌کنند.
* **برای کتابخانه‌ها**: معمولاً نیازی به افزودن Cargo. lock نیست، زیرا وابستگی‌ها توسط پروژه‌هایی که از آن کتابخانه استفاده می‌کنند، مدیریت می‌شوند.

### ****نتیجه‌گیری****

فایل Cargo. lock یکی از اجزای مهم مدیریت وابستگی‌ها در Rust است که به تضمین ثبات و پایداری پروژه کمک می‌کند. این فایل نسخه‌های دقیق هر وابستگی را ثبت کرده و اطمینان می‌دهد که ساخت‌های مکرر پروژه نتایج یکسانی تولید کنند.

**س8:**

**Rust Rover** یک **IDE تخصصی** برای زبان برنامه‌نویسی **Rust** است که توسط شرکت **Jet Brains** توسعه یافته است. این ابزار به‌طور خاص برای توسعه‌دهندگان Rust طراحی شده و شامل مجموعه‌ای از قابلیت‌ها و ویژگی‌های پیشرفته برای ساده‌سازی فرآیند توسعه، رفع اشکال، و بهبود بهره‌وری می‌باشد. Rust Rover بر پایه موتورهای قدرتمند Jet Brains، مانند IntelliJ، ساخته شده است و از ویژگی‌های خاص زبان Rust به طور کامل پشتیبانی می‌کند.

### ****ویژگی‌های اصلی Rust Rover****

#### 1. **تکمیل کد هوشمند (Smart Code Completion)**

Rust Rover با تحلیل دقیق کد منبع و وابستگی‌ها، پیشنهادهای هوشمندانه‌ای برای تکمیل کد ارائه می‌دهد که به نوشتن سریع‌تر و دقیق‌تر کد کمک می‌کند.

#### 2. **پشتیبانی از Cargo**

این IDE به‌طور یکپارچه از **Cargo**، ابزار مدیریت بسته و سیستم ساخت Rust، پشتیبانی می‌کند. کاربران می‌توانند پروژه‌ها را ایجاد، ساخت، و اجرا کرده و وابستگی‌ها را مدیریت کنند.

#### 3. **تجزیه و تحلیل ایستا (Static Analysis)**

Rust Rover از ابزارهای تحلیل ایستا برای شناسایی خطاهای احتمالی، هشدارها و بهینه‌سازی‌های کد پشتیبانی می‌کند. این ویژگی به بهبود کیفیت کد و جلوگیری از بروز باگ‌ها کمک می‌کند.

#### 4. **بازسازی کد (Refactoring)**

این IDE ابزارهای متنوعی برای بازسازی کد فراهم می‌کند، مانند تغییر نام متغیرها، توابع و کلاس‌ها، بدون شکستن منطق کد.

#### 5. **عیب‌یابی پیشرفته (Advanced Debugging)**

Rust Rover به ابزارهای پیشرفته‌ای برای **دیباگ کردن** کد مجهز است که به برنامه‌نویسان کمک می‌کند متغیرها و ساختارهای داده را به‌صورت زنده مشاهده کنند و مشکلات را سریع‌تر برطرف نمایند.

#### 6. **پشتیبانی از تست‌های خودکار**

این IDE به‌طور یکپارچه از اجرای تست‌ها با استفاده از **cargo test** پشتیبانی می‌کند و نتایج تست را به شکل بصری نمایش می‌دهد.

#### 7. **مستندسازی و ابزارهای کمک‌نویسنده**

Rust Rover به ابزارهای مستندسازی خودکار و بررسی قواعد نگارش کد مجهز است که کدنویسی استاندارد و قابل فهم را آسان‌تر می‌کند.

### ****مزایای Rust Rover****

* پشتیبانی قوی از زبان Rust و ابزارهای مرتبط مانند **Cargo** و **Clippy** .
* ادغام با سیستم‌های کنترل نسخه مانند **git** .
* محیط کاربری مدرن و قابل تنظیم.
* تجربه توسعه مشابه سایر IDEهای Jet Brains برای کاربران آشنا به IntelliJ IDEA.

### ****معایب و محدودیت‌ها****

* **نیاز به منابع سیستمی بیشتر** نسبت به ویرایشگرهای ساده‌تری مانند **VS Code** .
* **نسخه رایگان محدود**: ممکن است برخی قابلیت‌ها نیاز به اشتراک حرفه‌ای Jet Brains داشته باشند.

### ****نتیجه‌گیری****

Rust Rover یک IDE قدرتمند و حرفه‌ای برای توسعه‌دهندگان Rust است که با مجموعه‌ای از ابزارهای پیشرفته، فرآیند توسعه را سریع‌تر، ایمن‌تر و کارآمدتر می‌کند. این ابزار برای پروژه‌های بزرگ و توسعه‌دهندگانی که به ابزارهای پیشرفته نیاز دارند، انتخابی عالی محسوب می‌شود.

**س9:**

زبان **Rust** به دلیل ایمنی حافظه، عملکرد بالا، و قابلیت‌های همزمانی، در پروژه‌های متن‌باز و تجاری متعددی استفاده شده است. در ادامه به برخی از برنامه‌ها و پروژه‌های معروف که با Rust نوشته شده‌اند اشاره می‌کنیم:

### ****1. Firefox Quantum (بخش‌هایی از مرورگر فایرفاکس)****

* **توسعه‌دهنده**: Mozilla
* بخش‌های کلیدی مرورگر فایرفاکس، از جمله **موتور CSS با نام Servo**، با Rust بازنویسی شده‌اند که به بهبود کارایی و کاهش خرابی‌ها کمک کرده است.

### ****2. Ripgrep****

* **توسعه‌دهنده**: Andrew Gallant
* یک ابزار خط فرمان برای جستجوی سریع متن در فایل‌ها، مشابه grep اما بسیار سریع‌تر و کارآمدتر.
* **ویژگی‌ها**: سرعت بالا و پشتیبانی از تطبیق الگوهای پیچیده.

### ****3. Deno****

* **توسعه‌دهنده**: Ryan Dahl (خالق Node.js)
* یک محیط اجرایی امن برای جاوااسکریپت و تایپ‌اسکریپت که با Rust و V8 ساخته شده است.

### ****4. Bat****

* **توسعه‌دهنده**: sharkdp
* یک جایگزین زیبا و مدرن برای دستور cat با پشتیبانی از هایلایت کردن سینتکس و نمایش شماره خطوط.

### **5. Alacritty**

* **توسعه‌دهنده**: جامعه متن‌باز
* یک ترمینال سریع و مینیمال که از شتاب سخت‌افزاری GPU استفاده می‌کند.

### ****6. TiKV****

* **توسعه‌دهنده**: PingCAP
* یک پایگاه داده توزیع‌شده و سازگار با پروتکل‌های Key-Value که برای کاربردهای با تأخیر پایین طراحی شده است.

### ****7. Exa****

* **توسعه‌دهنده**: Benjamin Sago
* یک جایگزین مدرن برای دستور ls در لینوکس و یونیکس، با خروجی زیبا و ویژگی‌های اضافی.

### ****8. Meilisearch****

* **توسعه‌دهنده**: Meilisearch تیم
* یک موتور جستجوی سریع و متن‌باز برای برنامه‌های کاربردی مدرن.

### ****9. Tauri****

* **توسعه‌دهنده**: جامعه متن‌باز
* یک فریم‌ورک برای ایجاد برنامه‌های دسکتاپ که از Rust برای امنیت و عملکرد و WebView برای رابط کاربری استفاده می‌کند.

### ****10. OpenEthereum (قبلاً Parity)****

* **توسعه‌دهنده**: Parity Technologies
* یکی از کلاینت‌های محبوب برای بلاک‌چین اتریوم که با Rust نوشته شده است.

### ****نتیجه‌گیری****

Rust در پروژه‌های بسیاری استفاده می‌شود که عملکرد و ایمنی از اهمیت بالایی برخوردارند. این زبان همچنان به محبوبیت خود در حوزه‌های مختلفی مانند مرورگرها، پایگاه‌های داده، موتورهای جستجو، و ابزارهای سیستم‌عامل ادامه می‌دهد.

**س10**

زبان **Rust** به دلیل ایمنی بالا و استفاده بهینه از منابع، در زمینه **Concurrency (همزمانی)** و **Parallelism (موازی‌سازی)** بسیار قدرتمند است. در ادامه قابلیت‌های آن در این دو حوزه و یک مثال ساده توضیح داده شده‌اند.

### **1. همزمانی (Concurrency)**

Concurrency به معنای مدیریت چند کار به طور همزمان است، حتی اگر این کارها لزوماً به طور موازی اجرا نشوند. در Rust، مدل **مالکیت و وام‌گیری (Ownership & Borrowing)** ایمنی همزمانی را تضمین می‌کند. این زبان به کمک سیستم نوع خود تضمین می‌کند که داده‌ها بدون نیاز به قفل‌های صریح یا ابزارهای مدیریت دستی، به شکل ایمن بین نخ‌ها به اشتراک گذاشته شوند.

#### **ویژگی‌های کلیدی همزمانی در Rust**

* **استفاده از نخ‌ها (Threads):** کتابخانه استاندارد Rust از نخ‌ها با std::thread پشتیبانی می‌کند.
* **مالکیت انتقال داده بین نخ‌ها:** به اشتراک‌گذاری داده‌ها تنها زمانی مجاز است که ایمنی مالکیت برقرار باشد.
* **ارسال پیام با کانال‌ها (Channels):** std::sync::mpsc برای ارتباطات امن بین نخ‌ها استفاده می‌شود.

##### **مثال: ایجاد یک نخ ساده و اجرای تابع**

rust

Copy code

use std::thread;

fn main() {

let handle = thread::spawn(|| {

for i in 1..5 {

println!("Hello from spawned thread: {}", i);

}

});

// کد اصلی منتظر اجرای نخ بماند

handle.join().unwrap();

}

### **2. موازی‌سازی (Parallelism)**

Parallelism به معنای اجرای همزمان چندین عملیات به طور واقعی در چندین هسته پردازنده است. Rust ابزارهای متعددی برای موازی‌سازی ایمن ارائه می‌دهد.

#### **ویژگی‌های کلیدی موازی‌سازی در Rust**

* **کتابخانه Rayon:** برای اجرای موازی ساده و ایمن حلقه‌ها و داده‌های بزرگ. این کتابخانه از سیستم مالکیت Rust برای مدیریت ایمنی حافظه بهره می‌گیرد.
* **مجموعه‌های داده موازی (Parallel Iterators):** rayon به سادگی به Iteratorهای موازی تبدیل می‌شود.

##### **مثال: موازی‌سازی با Rayon**

rust

Copy code

use rayon::prelude::\*;

fn main() {

let numbers: Vec<i32> = (1..100).collect();

let sum: i32 = numbers.par\_iter().map(|x| x \* 2).sum();

println!("Sum of doubled numbers: {}", sum);

}

### **تفاوت اصلی بین Concurrency و Parallelism در Rust**

* **Concurrency** مربوط به مدیریت چندین کار به صورت همزمان است.
* **Parallelism** به معنای اجرای چند کار به طور همزمان روی چندین هسته پردازنده است.

### **ویژگی‌های برتر Rust در مدیریت همزمانی و موازی‌سازی**

1. **بدون شرط‌های رقابتی (Data Race-Free):** به لطف قوانین مالکیت، Rust کدهایی تولید می‌کند که از شرط‌های رقابتی مصون هستند.
2. **ایمنی زمان کامپایل:** بسیاری از مشکلات همزمانی قبل از اجرا شناسایی می‌شوند.
3. **کتابخانه‌های قدرتمند مانند Rayon و Crossbeam:** بهینه‌سازی برای کارهای موازی.

### **نتیجه‌گیری**

قابلیت‌های **Concurrency** و **Parallelism** در Rust به کمک مدل مالکیت و ابزارهای پیشرفته باعث ایمنی و کارایی بالا می‌شوند. Rust انتخابی عالی برای توسعه سیستم‌ها و برنامه‌های همزمان و موازی با حداقل خطاهای مرتبط با حافظه و نخ‌ها است.

**سوال11:**

زبان  **Rust** به دلیل ایمنی حافظه، سرعت بالا، و مدیریت بهینه منابع، به طور فزاینده‌ای برای **برنامه‌نویسی میکروکنترلرها** و **ربات‌ها** استفاده می‌شود. در این حوزه‌ها، Rust ویژگی‌هایی را ارائه می‌دهد که توسعه‌دهندگان را قادر می‌سازد کدی ایمن‌تر و کارآمدتر بنویسند. در ادامه قابلیت‌ها و ابزارهای موجود برای این منظور را بررسی می‌کنیم.

### ****1. Rust برای میکروکنترلرها****

Rust در برنامه‌نویسی **Embedded Systems** (سیستم‌های نهفته) برای میکروکنترلرها بسیار قدرتمند است.

#### **ویژگی‌های کلیدی**

* **مدیریت منابع بهینه:** بدون هزینه‌های اضافی (Zero-Cost Abstractions) برای کار با سخت‌افزار.
* **ایمنی حافظه بدون نیاز به Garbage Collector:** تضمین ایمنی حافظه با استفاده از مدل مالکیت.
* **مدیریت وقفه‌ها (Interrupt Handling):** با استفاده از ماکروهای ویژه مانند #[interrupt]، می‌توان به سادگی وقفه‌ها را مدیریت کرد.

#### **ابزارها و کتابخانه‌ها**

* **Nos td** : Rust قابلیت برنامه‌نویسی بدون کتابخانه استاندارد را دارد که برای سیستم‌های تعبیه‌شده ضروری است.
* **cortex-m** و **cortex-m-rt**: برای کار با میکروکنترلرهای ARM.
* **embedded-hal**: یک واسط استاندارد برای کار با سخت‌افزارهای مختلف.
* **rtic**: یک فریم‌ورک برای برنامه‌ریزی همزمان در سیستم‌های تعبیه‌شده.

##### **مثال ساده راه‌اندازی یک میکروکنترلر ARM**

rust

Copy code

#![no\_std]

#![no\_main]

use cortex\_m\_rt::entry;

use cortex\_m::peripheral::Peripherals;

#[entry]

fn main() -> ! {

let \_peripherals = Peripherals::take().unwrap();

loop {

// حلقه بی‌نهایت برای ادامه اجرای برنامه

}

}

### ****2. Rust برای رباتیک****

Rust به دلیل مدیریت ایمن نخ‌ها و کارایی بالا، گزینه‌ای مناسب برای ساخت سیستم‌های کنترلی پیچیده در ربات‌ها است.

#### **ویژگی‌های کلیدی**

* **Concurrency ایمن برای کنترل ربات‌ها**: Rust از مدل همزمانی بدون شرط‌های رقابتی استفاده می‌کند که برای کنترل همزمان موتورها و سنسورها بسیار مفید است.
* **کتابخانه‌های مخصوص رباتیک**:
  + **rust-ros**: برای کار با **Robot Operating System (ROS)**.
  + **robotics و pid-controller**: برای الگوریتم‌های کنترل و پردازش داده‌های سنسور.

### ****نمونه پروژه‌ها****

1. **MicroRust**: یک راهنما برای برنامه‌نویسی میکروکنترلرهای ARM با Rust.
2. **RoboWare و Tauri**: برای ایجاد سیستم‌های کنترلی رباتیک.
3. **بازی با ROS و Rust**: استفاده از Rust برای نوشتن نودهای ROS.

### ****مزایا برای برنامه‌نویسی میکروکنترلرها و ربات‌ها****

* **ایمنی حافظه و داده‌ها**: جلوگیری از باگ‌های خطرناک در سیستم‌های بلادرنگ.
* **مدیریت کم‌هزینه منابع**: مناسب برای سیستم‌های کم‌قدرت.
* **استفاده گسترده از کتابخانه‌های متن‌باز و جوامع فعال**.

### ****نتیجه‌گیری****

Rust با پشتیبانی از قابلیت‌های **Embedded Programming** و **Robotics**، ایمنی، کارایی و قابلیت‌های همزمانی را ارائه می‌دهد که آن را به گزینه‌ای عالی برای توسعه برنامه‌های میکروکنترلری و رباتیک تبدیل می‌کند. این زبان به طور مداوم در حال پیشرفت است و محبوبیت بیشتری در این حوزه‌ها پیدا می‌کند.

**سوال12**

در زبان **Rust**، مفهوم **Crate** به عنوان یک واحد پایه برای سازماندهی کد و مدیریت وابستگی‌ها به کار می‌رود. هر پروژه‌ی Rust یک یا چند **Crate** را شامل می‌شود که می‌تواند به صورت کتابخانه (Library) یا برنامه اجرایی (Binary) باشد.

### ****مفهوم Crate****

* یک **Crate** کوچک‌ترین واحد کامپایل در Rust است.
* کد Rust به فایل‌ها و ماژول‌ها سازماندهی می‌شود، اما کامپایلر Rust همیشه از Crate به عنوان واحد اصلی کامپایل استفاده می‌کند.
* هر Crate می‌تواند به سایر Crate‌ها وابسته باشد و این وابستگی‌ها با ابزار **Cargo** مدیریت می‌شوند.

### ****انواع Crate‌ها****

1. **Crate اجرایی (Binary Crate):**
   * یک برنامه مستقل قابل اجرا.
   * حاوی یک نقطه ورود (main function) است.
   * مثال:

rust

Copy code

fn main() {

println!("Hello, Rust!");

}

1. **Crate کتابخانه (Library Crate):**
   * شامل کدهای قابل استفاده مجدد بدون نقطه ورود مشخص.
   * معمولاً فایل lib.rs در پوشه src دارد.
   * مثال:

rust

Copy code

pub fn greet(name: &str) {

println!("Hello, {}!", name);

}

### ****ایجاد و استفاده از Crate‌ها****

#### **ایجاد یک Crate جدید با Cargo**

bash

Copy code

cargo new my\_crate

* این دستور یک پروژه Rust با فایل‌های Cargo.toml و src/main.rs ایجاد می‌کند.

#### **اضافه کردن وابستگی‌ها**

با افزودن Crate‌های دیگر به فایل Cargo.toml، می‌توان وابستگی‌ها را مدیریت کرد:

toml

Copy code

[dependencies]

rand = "0.8"

سپس می‌توان از این Crate استفاده کرد:

rust

Copy code

use rand::Rng;

fn main() {

let random\_number = rand::thread\_rng().gen\_range(1..10);

println!("Random number: {}", random\_number);

}

### ****تفاوت Crate و Module****

| **Crate** | **Module** |
| --- | --- |
| واحد اصلی کامپایل و لینک‌دهی. | برای سازماندهی کد درون یک Crate استفاده می‌شود. |
| می‌تواند Binary یا Library باشد. | در یک Crate تعریف می‌شود و سلسله‌مراتبی از کدها ایجاد می‌کند. |

### ****جمع‌بندی****

Crate‌ها قلب سیستم بسته‌بندی و کامپایل در Rust هستند. استفاده از Crate‌ها به توسعه‌دهندگان کمک می‌کند تا کد خود را به اجزای کوچک‌تر و قابل استفاده مجدد تقسیم کنند و از کتابخانه‌های متن‌باز Rust به سادگی بهره ببرند

سوال13::

در زبان **Rust**، **Iterator** و **Generator** هر دو برای تولید مقادیر به صورت دنباله‌ای به کار می‌روند، اما تفاوت‌های مهمی در نحوه عملکرد و استفاده از آن‌ها وجود دارد.

### **1. Iterator**

یک **Iterator** در Rust به شما امکان می‌دهد روی مجموعه‌ای از داده‌ها یا مقادیری که به صورت متوالی تولید می‌شوند، تکرار کنید.

#### **ویژگی‌ها**

* یک Iterator با متد next مقادیر را یکی‌یکی باز می‌گرداند.
* پس از تمام شدن مقادیر، next مقدار None بازمی‌گرداند.
* از قبل تمام مقادیر در مجموعه‌ای ذخیره شده‌اند یا در حین اجرا تولید می‌شوند.

#### **مثال ساده از Iterator**

rust

Copy code

fn main() {

let numbers = vec![1, 2, 3, 4];

let mut iter = numbers.iter();

while let Some(num) = iter.next() {

println!("{}", num);

}

}

در این مثال، iter یک Iterator است که روی عناصر بردار (vec) پیمایش می‌کند.

### **2. Generator**

**Generator** (تولیدکننده) ساختاری است که مقادیر را به صورت تدریجی و در واکنش به درخواست تولید می‌کند، بدون اینکه نیاز به داشتن تمام داده‌ها به صورت همزمان باشد. در Rust، این مفهوم به‌طور ضمنی توسط ویژگی **async/await** و یا قابلیت‌های جدیدتر مرتبط با **Generators** ارائه شده است.

#### **ویژگی‌ها**

* Generators با استفاده از yield مقادیر را تولید می‌کنند.
* به جای برگشت مستقیم مقادیر، جریان محاسبات را به نقطه بعدی متوقف می‌کنند.
* حالت داخلی خود را بین فراخوانی‌ها حفظ می‌کنند.

##### **مثال ساده (با استفاده از کتابخانه خارجی** generator**)**

rust

Copy code

use generator::Gn;

fn main() {

let mut gen = Gn::new\_scoped(|mut yielder| {

for i in 0..5 {

yielder.yield\_(i);

}

});

while let Some(value) = gen.next() {

println!("Generated value: {}", value);

}

}

در این مثال، هر بار که gen.next() فراخوانی می‌شود، Generator مقدار بعدی را تولید می‌کند.

### **تفاوت‌های کلیدی**

| **ویژگی** | **Iterator** | **Generator** |
| --- | --- | --- |
| **مدل تولید داده‌ها** | مقادیر را به صورت متوالی از قبل آماده می‌کند یا می‌خواند | مقادیر را به صورت تنبل و به درخواست تولید می‌کند |
| **حفظ حالت داخلی** | حالت داخلی ندارد (به جز شمارنده داخلی) | حالت داخلی را بین تولید مقادیر حفظ می‌کند |
| **کلمه کلیدی** | از next برای دریافت مقدار بعدی استفاده می‌شود | از yield برای توقف و از سرگیری استفاده می‌شود |

### **جمع‌بندی**

* **Iterator** برای پیمایش در مجموعه‌ها و تولید مقادیر از قبل تعریف شده مناسب است.
* **Generator** برای تولید مقادیر به صورت پویا و با حفظ حالت در حین تولید مفید است.

تفاوت عمده در نحوه مدیریت حالت و زمان تولید مقادیر است که Generators را در زمینه‌هایی مانند **پروگرام‌نویسی async** یا **محاسبات پیچیده و بلادرنگ** قدرتمندتر می‌کند

سوال14::

در زبان **Rust**، مفهوم **تک‌شکلی (Polymorphism)** به معنای توانایی نوشتن کدهای عمومی است که می‌توانند با انواع مختلف داده کار کنند. Rust از **تک‌شکلی ایستا (Static Polymorphism)** و **تک‌شکلی پویا (Dynamic Polymorphism)** پشتیبانی می‌کند. این دو نوع به ترتیب از **جنریک‌ها (Generics)** و **صفات (Traits)** استفاده می‌کنند.

### **1. تک‌شکلی ایستا با Generics**

تک‌شکلی ایستا به این معناست که نوع داده‌ها در زمان کامپایل مشخص می‌شود. جنریک‌ها به شما امکان می‌دهند که کدی بنویسید که برای انواع مختلف بدون تکرار قابل استفاده باشد.

#### **مثال ساده: استفاده از Generics**

rust

Copy code

fn print\_value<T: std::fmt::Debug>(value: T) {

println!("{:?}", value);

}

fn main() {

print\_value(42); // با یک عدد صحیح

print\_value("Rust"); // با یک رشته

}

در این مثال، T یک نوع جنریک است که می‌تواند هر نوعی باشد، مشروط به اینکه صفت Debug را پیاده‌سازی کرده باشد.

### **2. تک‌شکلی پویا با Traits**

تک‌شکلی پویا به معنای امکان استفاده از **صفات (Traits)** برای انتزاعی کردن رفتارهایی است که انواع مختلف می‌توانند پیاده‌سازی کنند. این نوع تک‌شکلی با استفاده از اشاره‌گرها یا جعبه‌های هوشمند مانند Box<dyn Trait> انجام می‌شود.

#### **مثال ساده: استفاده از Traits**

rust

Copy code

trait Animal {

fn speak(&self);

}

struct Dog;

impl Animal for Dog {

fn speak(&self) {

println!("Woof!");

}

}

struct Cat;

impl Animal for Cat {

fn speak(&self) {

println!("Meow!");

}

}

fn make\_animal\_speak(animal: &dyn Animal) {

animal.speak();

}

fn main() {

let dog = Dog;

let cat = Cat;

make\_animal\_speak(&dog); // Woof!

make\_animal\_speak(&cat); // Meow!

}

در این مثال، Animal یک Trait است که رفتار مشترک speak را تعریف می‌کند. توابع می‌توانند به صورت پویا با هر نوعی که این Trait را پیاده‌سازی کرده باشد، کار کنند.

### **تفاوت‌های کلیدی بین تک‌شکلی ایستا و پویا در Rust**

| **ویژگی** | **تک‌شکلی ایستا (Generics)** | | **تک‌شکلی پویا (Traits)** |
| --- | --- | --- | --- |
| **زمان تعیین نوع** | در زمان کامپایل | در زمان اجرا | |
| **کارایی** | سریع‌تر (بدون سربار) | کندتر (به دلیل سربار پویا) | |
| **استفاده از کلمه کلیدی** | T یا هر نام جنریک دیگری | dyn Trait برای پویایی | |

### **نتیجه‌گیری**

تک‌شکلی در Rust از طریق **Generics** و **Traits** مدیریت می‌شود و به توسعه‌دهندگان امکان می‌دهد کدهای انعطاف‌پذیر، ایمن و قابل استفاده مجدد بنویسند. این ترکیب از تک‌شکلی ایستا و پویا یکی از ویژگی‌های کلیدی Rust برای کارایی بالا و ایمنی نوع است

سوال15::

**مالکیت (Ownership)** یکی از مفاهیم اساسی و منحصربه‌فرد در زبان **Rust** است که ایمنی حافظه را بدون نیاز به مدیریت دستی یا استفاده از جمع‌آوری زباله (Garbage Collector) فراهم می‌کند. این مدل، قوانین مشخصی برای مدیریت حافظه و عمر متغیرها دارد. در ادامه، مفهوم مالکیت و قوانین آن توضیح داده می‌شود.

### **قوانین مالکیت در Rust**

1. **هر مقدار یک مالک (Owner) دارد.**
2. **در هر لحظه فقط یک مالک می‌تواند به یک مقدار دسترسی داشته باشد.**
3. **زمانی که مالک از بین می‌رود، مقدار نیز آزاد می‌شود.**

### **انتقال مالکیت (Move)**

زمانی که یک مقدار به متغیر جدیدی اختصاص داده می‌شود، مالکیت به آن منتقل می‌شود و متغیر اصلی دیگر به مقدار دسترسی ندارد.

#### **مثال**

rust

Copy code

fn main() {

let s1 = String::from("Hello");

let s2 = s1; // مالکیت به s2 منتقل می‌شود

// println!("{}", s1); // خطا: s1 دیگر مالک نیست

}

در اینجا، s1 دیگر به مقدار اشاره نمی‌کند زیرا مالکیت به s2 منتقل شده است.

### **وام‌گیری (Borrowing)**

به جای انتقال مالکیت، می‌توان از وام‌گیری استفاده کرد که به دو صورت انجام می‌شود:

* **وام‌گیری غیرقابل تغییر (**&T**)**: چندین وام‌گیرنده می‌توانند به داده دسترسی داشته باشند.
* **وام‌گیری قابل تغییر (**&mut T**)**: فقط یک وام‌گیرنده مجاز است.

#### **مثال**

rust

Copy code

fn print\_message(message: &String) {

println!("{}", message);

}

fn main() {

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

print\_message(&s); // وام‌گیری بدون انتقال مالکیت

println!("{}", s); // همچنان قابل استفاده

}

### **وام‌گیری متغیر**

وام‌گیری متغیر فقط یک بار در هر زمان مجاز است.

#### **مثال**

rust

Copy code

fn change\_message(message: &mut String) {

message.push\_str(", Rust!");

}

fn main() {

let mut s = String::from("Hello");

change\_message(&mut s); // وام‌گیری متغیر

println!("{}", s); // مقدار تغییر یافته

}

### **قابلیت وام‌گیری در قوانین ایمنی حافظه**

* **وام‌گیری متغیر و ثابت نمی‌توانند همزمان اتفاق بیفتند.**
* **تعداد زیادی وام ثابت یا یک وام متغیر در هر لحظه مجاز است.**

### **نتیجه‌گیری**

مدل مالکیت در Rust ایمنی حافظه را در زمان کامپایل تضمین می‌کند و مانع بروز مشکلاتی مانند دسترسی به حافظه آزادشده یا شرط‌های رقابتی می‌شود. این رویکرد یکی از دلایل اصلی انتخاب Rust برای توسعه سیستم‌های ایمن و کارآمد است

**سوال16::**

**Borrowing** یکی از ویژگی‌های کلیدی **مدل مالکیت (Ownership)** در زبان **Rust** است که امکان دسترسی به داده‌ها بدون انتقال مالکیت را فراهم می‌کند. این ویژگی از طریق ارجاع‌ها (**References**) به کار گرفته می‌شود و ایمنی حافظه را در زمان کامپایل تضمین می‌کند. Borrowing به دو صورت وجود دارد: **وام‌گیری غیرقابل تغییر** و **وام‌گیری قابل تغییر**.

### **1. وام‌گیری غیرقابل تغییر (Immutable Borrowing)**

در وام‌گیری غیرقابل تغییر، می‌توان از داده بدون تغییر آن استفاده کرد.

#### **مثال وام‌گیری غیرقابل تغییر**

rust

Copy code

fn print\_message(message: &String) {

println!("{}", message);

}

fn main() {

let s = String::from("Hello, Rust!");

print\_message(&s); // وام‌گیری غیرقابل تغییر

println!("{}", s); // همچنان می‌توان از s استفاده کرد

}

در این مثال:

* &s یک ارجاع غیرقابل تغییر به s است.
* مالکیت s منتقل نمی‌شود و همچنان می‌توان پس از وام‌گیری از آن استفاده کرد.

### **2. وام‌گیری قابل تغییر (Mutable Borrowing)**

وام‌گیری قابل تغییر به شما امکان می‌دهد مقدار متغیر را تغییر دهید، اما در هر لحظه تنها یک وام‌گیری قابل تغییر مجاز است.

#### **مثال وام‌گیری قابل تغییر**

rust

Copy code

fn change\_message(message: &mut String) {

message.push\_str(" Learning Rust is fun!");

}

fn main() {

let mut s = String::from("Hello, Rust");

change\_message(&mut s); // وام‌گیری قابل تغییر

println!("{}", s); // مقدار تغییر یافته

}

در اینجا:

* &mut s یک ارجاع قابل تغییر است.
* تابع change\_message مقدار رشته s را تغییر می‌دهد.

### **قوانین Borrowing**

1. می‌توان هر تعداد ارجاع غیرقابل تغییر داشت.
2. فقط یک ارجاع قابل تغییر در هر زمان مجاز است.
3. ارجاع‌های غیرقابل تغییر و قابل تغییر نمی‌توانند همزمان وجود داشته باشند.

#### **مثال نقض قوانین Borrowing**

rust

Copy code

fn main() {

let mut s = String::from("Rust");

let r1 = &s; // ارجاع غیرقابل تغییر

let r2 = &mut s; // ارجاع قابل تغییر

println!("{}", r1); // خطا: همزمانی ارجاع‌های متضاد

}

### **نتیجه‌گیری**

ویژگی Borrowing در Rust به شما امکان می‌دهد کدهای ایمن‌تری بنویسید و از مشکلاتی مانند شرط‌های رقابتی یا دسترسی به حافظه آزادشده جلوگیری کنید. این ویژگی با تضمین ایمنی در زمان کامپایل یکی از مزایای مهم Rust است

**سوال17::**

**استنتاج نوع (Type Inference)** در زبان **Rust** به این معناست که کامپایلر می‌تواند نوع متغیرها و مقادیر را بدون نیاز به اعلام صریح نوع توسط برنامه‌نویس تعیین کند. Rust با استفاده از قوانین استنتاج نوع، به ایمنی و سادگی کد کمک می‌کند.

### ****1. استنتاج نوع متغیر****

وقتی نوع متغیر به‌صراحت مشخص نشده باشد، کامپایلر Rust از مقدار انتساب‌شده برای تعیین نوع استفاده می‌کند.

#### **مثال**

rust

Copy code

fn main() {

let x = 42; // کامپایلر نوع x را به i32 استنتاج می‌کند

let y = 3.14; // نوع y به f64 استنتاج می‌شود

println!("x = {}, y = {}", x, y);

}

در این مثال، x به i32 و y به f64 استنتاج شده‌اند.

### ****2. استنتاج نوع برای توابع و جنریک‌ها****

کامپایلر Rust می‌تواند نوع آرگومان‌های تابع یا مقدار بازگشتی را بر اساس نحوه استفاده تعیین کند.

#### **مثال تابع**

rust

Copy code

fn add(a: i32, b: i32) -> i32 {

a + b

}

fn main() {

let sum = add(10, 20); // کامپایلر نوع sum را i32 تعیین می‌کند

println!("Sum is {}", sum);

}

نوع sum به صورت خودکار i32 استنتاج می‌شود.

### ****3. استنتاج در جنریک‌ها****

جنریک‌ها به‌طور معمول نیاز به اعلان صریح ندارند و کامپایلر نوع را بر اساس استفاده استنتاج می‌کند.

#### **مثال جنریک**

rust

Copy code

fn print\_value<T: std::fmt::Debug>(value: T) {

println!("{:?}", value);

}

fn main() {

print\_value(100); // نوع T به i32 استنتاج می‌شود

print\_value("Rust"); // نوع T به &str استنتاج می‌شود

}

### ****4. استنتاج نوع در عملیات****

کامپایلر نوع را بر اساس اپراتورها و استفاده از متغیرها تعیین می‌کند.

#### **مثال**

rust

Copy code

fn main() {

let a = 5;

let b = 2.5;

let c = a as f64 + b; // تبدیل a به f64، c به f64 استنتاج می‌شود

println!("c = {}", c);

}

### ****5. محدودیت‌های استنتاج نوع****

گاهی کامپایلر نیاز به اطلاعات اضافی دارد، و در این موارد باید نوع به‌صورت صریح تعیین شود.

#### **مثال عدم استنتاج**

rust

Copy code

fn main() {

let guess: u32 = "42".parse().expect("Not a number!");

}

بدون u32، کامپایلر نمی‌تواند تعیین کند چه نوع عددی باید استفاده شود.

### ****جمع‌بندی****

استنتاج نوع در Rust کد را ساده‌تر و خواناتر می‌کند، در حالی که همچنان ایمنی نوع را تضمین می‌کند. با این حال، وقتی اطلاعات کافی برای استنتاج وجود نداشته باشد، اعلان صریح نوع ضروری است

سوال18::

در زبان **Rust**، **اشاره‌گرهای هوشمند (Smart Pointers)** نوعی از ساختارهای داده هستند که علاوه بر نگهداری آدرس حافظه، ویژگی‌ها و قابلیت‌های اضافی مانند مدیریت مالکیت، شمارش مراجع، یا تخصیص و آزادسازی خودکار حافظه را ارائه می‌دهند. برخلاف اشاره‌گرهای خام در زبان‌هایی مانند C یا C++، اشاره‌گرهای هوشمند در Rust از قوانین مالکیت و ایمنی حافظه پیروی می‌کنند و به جلوگیری از مشکلات رایج در مدیریت حافظه کمک می‌کنند.

### **ویژگی‌های کلیدی اشاره‌گرهای هوشمند در Rust**

1. **مالکیت و مدیریت حافظه خودکار**: حافظه به‌صورت خودکار هنگام خروج از دامنه آزاد می‌شود.
2. **استفاده از صفات مانند** Deref **و** Drop: برای ارائه رفتارهای مشابه اشاره‌گر و پاک‌سازی خودکار.
3. **ایمنی حافظه**: مانع دسترسی به حافظه آزادشده یا وضعیت‌های رقابتی می‌شود.

### **انواع اصلی اشاره‌گرهای هوشمند در Rust**

#### 1. **Box<T>**

Box ساده‌ترین اشاره‌گر هوشمند است که داده‌ها را روی هیپ ذخیره می‌کند. مالکیت داده‌ها را در اختیار دارد و هنگام خروج از دامنه حافظه را آزاد می‌کند.

##### **مثال Box<T>**

rust

Copy code

fn main() {

let b = Box::new(5); // عدد 5 روی هیپ ذخیره می‌شود

println!("b = {}", b);

}

#### 2. **Rc<T> (Reference Counting)**

Rc<T> به شما امکان می‌دهد چندین مالک برای یک مقدار در زمان کامپایل داشته باشید. این اشاره‌گر برای اشتراک‌گذاری داده‌ها در محیط‌های تک‌نخی استفاده می‌شود.

##### **مثال Rc<T>**

rust

Copy code

use std::rc::Rc;

fn main() {

let a = Rc::new(5); // مقدار 5 با شمارش مرجع

let b = Rc::clone(&a);

println!("a = {}, count = {}", a, Rc::strong\_count(&a));

}

#### 3. **Arc<T> (Atomic Reference Counting)**

Arc<T> مشابه Rc<T> است، اما برای محیط‌های چندنخی استفاده می‌شود و ایمنی در برابر شرایط رقابتی را فراهم می‌کند.

#### 4. **RefCell<T>**

RefCell<T> اجازه می‌دهد قوانین وام‌گیری (Borrowing) به‌صورت پویا بررسی شوند. این ساختار امکان وام‌گیری متغیر در زمان اجرا را فراهم می‌کند.

##### **مثال RefCell<T>**

rust

Copy code

use std::cell::RefCell;

fn main() {

let data = RefCell::new(5);

\*data.borrow\_mut() += 1;

println!("data = {}", data.borrow());

}

### **جمع‌بندی**

اشاره‌گرهای هوشمند در Rust ابزاری قدرتمند برای مدیریت ایمن حافظه هستند و مزایای ایمنی حافظه را بدون هزینه مدیریت دستی فراهم می‌کنند. انتخاب نوع مناسب اشاره‌گر به شرایط مالکیت و هم‌زمانی در برنامه بستگی دارد

سوال 19::

در زبان **Rust**، **Trait** مشابه **رابط‌ها (Interfaces)** در سایر زبان‌های برنامه‌نویسی است. Trait مجموعه‌ای از توابع و رفتارهایی را تعریف می‌کند که یک نوع می‌تواند آن‌ها را پیاده‌سازی کند. استفاده از Trait به شما امکان می‌دهد که رفتارهای مشترک را برای انواع مختلف تعریف کرده و از **چندریختی (Polymorphism)** بهره‌مند شوید.

### ****تعریف یک Trait****

یک Trait با استفاده از کلمه کلیدی trait تعریف می‌شود. سپس هر نوعی که این Trait را پیاده‌سازی کند، باید تمام توابع مشخص‌شده در Trait را تعریف کند.

#### **مثال ساده**

rust

Copy code

trait Greet {

fn say\_hello(&self);

}

struct Person {

name: String,

}

impl Greet for Person {

fn say\_hello(&self) {

println!("Hello, my name is {}!", self.name);

}

}

fn main() {

let person = Person {

name: String::from("Alice"),

};

person.say\_hello(); // فراخوانی متد تعریف‌شده در Trait

}

در این مثال:

* Greet یک Trait است که تابع say\_hello را تعریف می‌کند.
* ساختار Person این Trait را پیاده‌سازی کرده است.
* می‌توان متد say\_hello را برای نمونه‌های Person فراخوانی کرد.

### ****استفاده از Trait در توابع****

می‌توان از Trait به‌عنوان محدودیت برای جنریک‌ها استفاده کرد تا تنها نوع‌هایی که آن Trait را پیاده‌سازی می‌کنند، مجاز باشند.

#### **مثال با Trait Bound**

rust

Copy code

trait Area {

fn area(&self) -> f64;

}

struct Circle {

radius: f64,

}

impl Area for Circle {

fn area(&self) -> f64 {

3.14 \* self.radius \* self.radius

}

}

fn print\_area<T: Area>(shape: T) {

println!("The area is {}", shape.area());

}

fn main() {

let circle = Circle { radius: 5.0 };

print\_area(circle);

}

در این مثال، فقط نوع‌هایی که Trait Area را پیاده‌سازی کرده‌اند می‌توانند به تابع print\_area ارسال شوند.

### ****جمع‌بندی****

* **Trait** ابزاری برای تعریف رفتارهای مشترک است.
* پیاده‌سازی Trait برای انواع مختلف امکان‌پذیر است.
* استفاده از Traits باعث افزایش انعطاف‌پذیری و استفاده مجدد از کد می‌شود

**سوال20::**

در زبان **Rust**، **بستن (Closure)** نوعی تابع ناشناس است که می‌تواند متغیرهای محیطی خود را **گرفته (capture)** و از آن‌ها استفاده کند. بستن‌ها شبیه **لامبداها (Lambdas)** در زبان‌های دیگر هستند و با استفاده از نماد |...| برای تعریف آرگومان‌ها و {...} برای بدنه نوشته می‌شوند.

### ****تعریف Closure****

ساختار کلی یک بستن به این شکل است:

rust

Copy code

let closure = |parameter1, parameter2| {

// بدنه بستن

};

### ****مثال ساده از Closure****

rust

Copy code

fn main() {

let add = |a: i32, b: i32| a + b; // بستن که دو عدد را جمع می‌کند

let result = add(5, 3);

println!("Result is: {}", result); // خروجی: Result is: 8

}

در این مثال، بستن add دو آرگومان می‌گیرد و آن‌ها را جمع می‌کند.

### ****بستن با گرفتن متغیرهای محیطی****

بستن‌ها می‌توانند متغیرهای خارج از دامنه خود را گرفته و به آن‌ها دسترسی داشته باشند.

#### **مثال گرفتن متغیر محیطی**

rust

Copy code

fn main() {

let x = 10;

let add\_to\_x = |y| x + y; // بستن که از x استفاده می‌کند

let result = add\_to\_x(5);

println!("Result is: {}", result); // خروجی: Result is: 15

}

### ****سه نوع Capture در Closure****

1. **By Reference (&T)**: بستن متغیرها را با ارجاع می‌گیرد.
2. **By Mutable Reference (&mut T)**: بستن متغیرها را با ارجاع متغیر می‌گیرد.
3. **By Value (T)**: بستن مالکیت متغیرها را می‌گیرد.

#### **مثال Capture By Value**

rust

Copy code

fn main() {

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

let consume = || println!("{}", s); // بستن مالکیت s را می‌گیرد

consume();

// println!("{}", s); // خطا: s منتقل شده و دیگر قابل دسترسی نیست

}

### ****Closure به‌عنوان آرگومان تابع****

بستن‌ها می‌توانند به‌عنوان آرگومان برای توابع استفاده شوند.

rust

Copy code

fn apply<F>(f: F)

where

F: FnOnce() {

f();

}

fn main() {

let greeting = || println!("Hello, Rust!");

apply(greeting);

}

در اینجا FnOnce نوع Trait است که بستن‌ها را مدیریت می‌کند.

### ****جمع‌بندی****

بستن‌ها در Rust ابزار قدرتمندی برای نوشتن کدهای مختصر و انعطاف‌پذیر هستند. آن‌ها می‌توانند متغیرهای محیطی را گرفته و بسته به نیاز به‌صورت ارجاع یا مالکیت عمل کنند

**سوال21::**

در زبان **Rust**، هدف اصلی ایمنی حافظه و جلوگیری از خطاهای رایج برنامه‌نویسی مانند دسترسی به حافظه آزادشده یا شرط‌های رقابتی است. با این حال، برخی مواقع نیاز به کدهای سطح پایین یا عملیات ناامن (مانند دسترسی مستقیم به اشاره‌گرها) وجود دارد. برای چنین مواردی، Rust به شما اجازه می‌دهد **کدهای ناامن (Unsafe Code)** را بنویسید، اما این کدها در یک بلوک خاص مشخص می‌شوند و مسئولیت ایمنی به عهده برنامه‌نویس خواهد بود.

### ****کد ناامن در Rust****

کدهای ناامن با کلمه کلیدی unsafe تعریف می‌شوند. این کدها به شما اجازه می‌دهند عملیاتی را انجام دهید که قوانین ایمنی Rust را نقض می‌کنند، مانند:

* دسترسی به اشاره‌گرهای خام (\*const T و \*mut T)
* فراخوانی توابع ناامن
* تغییر متغیرهای ایستا (global mutable variables)

### ****مثال ساده از استفاده Unsafe****

#### **دسترسی به اشاره‌گر خام**

rust

Copy code

fn main() {

let mut num = 5;

let r1 = &num as \*const i32; // اشاره‌گر خام خواندنی

let r2 = &mut num as \*mut i32; // اشاره‌گر خام نوشتنی

unsafe {

println!("r1 is: {}", \*r1);

\*r2 = 10;

println!("r2 is now: {}", \*r2);

}

}

در این مثال:

* r1 و r2 اشاره‌گرهای خام هستند که بدون بررسی قوانین مالکیت استفاده می‌شوند.
* استفاده از این اشاره‌گرها تنها در داخل بلوک unsafe مجاز است.

### ****فراخوانی توابع ناامن****

rust

Copy code

unsafe fn dangerous\_function() {

println!("This is a dangerous function.");

}

fn main() {

unsafe {

dangerous\_function(); // فراخوانی تابع ناامن

}

}

### ****زمان استفاده از Unsafe****

استفاده از کدهای ناامن تنها زمانی مجاز است که نیاز به:

* **تعامل با کدهای سطح پایین یا زبان‌های دیگر** (مانند C)
* **دستیابی به بهینه‌سازی‌های خاص** که در حالت ایمن قابل دستیابی نیست
* **پیاده‌سازی الگوریتم‌های پیچیده** که با قوانین معمول Rust سازگار نیستند

### ****جمع‌بندی****

Rust اجازه استفاده از کد ناامن را می‌دهد اما به صراحت از آن‌ها می‌خواهد در بلوک‌های مشخصی استفاده شوند. این ویژگی کمک می‌کند تا کدهای ایمن و ناامن از یکدیگر جدا شوند و مسئولیت ایمنی کد ناامن به عهده برنامه‌نویس باشد.

سوال22::

زبان **Rust** به**حافظه، کارایی بالا، و مدیریت مالکیت** بدون نیاز به جمع‌آوری**توسعه قراردادهای هوشمند** تبدیل شده است. برنامه های هوشمند برنامه هایی هستند که بر روی بلوک

### **چرا Rust برای قراردادهای هوشمند مناسب است؟**

1. **حافظه ایمنی** : زنگ از مشکلات مربوط به اشاره گرهای غیرمعتبر و دسترسی به حافظه آزاد شده جلوگیری می کند.
2. **بدون نیاز به زباله گردآوری** : این ویژگی باعث کاهش می شود
3. **مدیریت دقیق مالکیت و منابع** : Rust با سیستم مالکیت خود از شرایط رقابتی مانع ایجاد و امنیت بیشتر می شود
4. **کارایی بالا** : اجرای سریع کدها باعث بهبود عملکرد بلوک‌چین‌ می‌شود که از قرارداد

### **کاربردهای Rust در پلتفرم‌های قرارداد هوشمند**

#### 1. **سولانا**

* **Solana** یکی از بلاک‌چین‌های پرسرعت است که از Ru
* قراردادهای هوشمند در Solana به نام **برنامه‌ها (برنامه‌ها)** شناخته می‌شوند.

**نمونه کد قرارداد هوشمند در Solana با Rust** :

زنگ زدگی

کد را کپی کنید

use solana\_program::{

account\_info::AccountInfo,

entrypoint,

entrypoint::ProgramResult,

pubkey::Pubkey,

};

entrypoint!(process\_instruction);

account\_info::AccountInfo,

entrypoint,

entrypoint::ProgramResult,

pubkey::Pubkey,

};

entrypoint!(process\_i

account\_info::AccountInfo,

entrypoint,

entrypoint::Pro

pub fn process\_instruction(

\_program\_id: &Pubkey,

\_accounts: &[AccountInfo],

\_instruction\_data: &[

\_program\_id: &Pu

u8],

) -> ProgramResult {

msg!("Hello, Solana!");

Ok(())

}

#### 2. **Polkadot/Substrate**

* **بستر** یک فریم‌ورک برا
* قراردادهای هوشمند در Substrate با **ink!** نوشته می‌شوند، که یک
* **نمونه کد قرارداد هوشمند با ink!** :

زنگ زدگی

کد را کپی کنید

#[ink::contract]

mod my\_contract {

#[ink(storage)]

pub struct MyContract {

value:

value:

u32,

}

}

impl MyContract {

#[ink(constructor)]

pub fn new(init\_value: u32) -> Self {

Self { value: init\_value }

}

#[ink(message)]

pub fn get\_value(&self) -> u32 {

self.value

}

}

}

### **ویژگی‌های خاصبه‌فرد Rust برای قراردادهای هوشمند**

* **سیستم نوع قوی** : از خیلی از
* **پشتیبانی از برنامه نویسی همزمان و ناهمزمان** : مفید برای پردازش های پیچیده

### **جمع بندی**

Rust به ویژگی‌های ایمنی و کارایی، به یکی از زبان‌های کلیدی برای توسعه قراردادهای هوشمند در پلتفرم‌های پیشرفته مانند Solana و Polkadot تبدیل می‌شود. این زبان می‌تواند به سمت توسعه‌دهندگان را به ایجاد سیستم‌های غیرمتمرکز ایمان‌

سوال23:

زبان **Rust** به**پلتفرم های بلوکچین** و \*\*قراردادهای**قراردادهای هوشمند** پیدا کرده است. Rust با ترکیب \*\*ایمنی حافظه، کارایی بالا، مدیریت بدون Garbage Collect**حافظه، کارایی بالا، مدیریت بدون زباله جمع آوری، و پشتیبانی از برنامه نویسی همزمان** ، گزینه های ایده آل برای ساختارهای پیچیده و تو

### **ویژگی های کلیدی Rust برای توسعه بلوکچین**

1. **حافظه امن** : سیستم مالکیت و وام‌گیری (مالکیت و قرض گرفتن) از مشکلات رایج در مدیریت جلوگیری از می‌کند.
2. **کارایی بالا** : کدهای کام
3. **بدون نیاز به Garbage Collection** : مناسب برای برنامه های ح
4. **برنامه‌نویسی هم‌زمان ایمان** : Rust ابزارهایی مانند tokioو asyncبرای مدیریت ایمن کدهای همزمان فر

### **کاربردهای Rust در بلوک‌چین**

#### 1. **ساخت بلوک‌چین‌های سفارشی**

پلتفرم‌های بلوک‌چین جدید و سفارشی از Rust برای ساخت نودها و اجزای مختلف خود استف

* **Polkadot** : یکی از مشهورترین پروژه‌های بلوک‌چین س
* **بستر**: فریم‌ورکی برای ساخت بلوک‌چین‌های سفارشی است که کاملاً با Rust پیاده‌سازی شده است.

**نمونه کد ساده برای زیرلایه:**

زنگ زدگی

کد را کپی کنید

use sp\_runtime::traits::{BlakeTwo256, Block as BlockT, Header as HeaderT};

pub struct MyBlockchain;

impl BlockT for MyBlockchain {

type Header = HeaderT<BlakeTwo256>;

}

#### 2. **توسعه قراردادهای هوشمند**

بلوک‌چین‌هایی مانند **Solana** و **Polkadot** از Rust برای توسعه قراردادهای هوشمند استفاده می‌کنند:

* **Solana** به توسعه‌دهندگان Rust برای ایجاد برنامه‌های غیرمتمرکز سریع و کم‌ه
* **جوهر!** برای نوشتن قراردادهای هوشمند در Substrate استفاده می‌شود.

#### 3. **پیاده‌سازی الگوریتم‌های جماع**

Rust به دلیل کارایی بالا در پیاده‌سازی الگوریتم‌های پیچیده جماع مانند **Proof of Stake (PoS)** ، **Byzantine Fault Tolerance (BFT)** و \*\*Proof of Authority**Proof of Authority (PoA)** مورد استفاده قرار می‌گیرد.

### **پروژه های بزرگ بلوکچین با Rust**

* **Polkadot** : یک شبکه بلوک‌چین چندزنجیره‌ای که از Substrate برای ساخت و اتصال بلوک‌چین‌ها استفاده می‌کند.
* **Solana** : یکی از سریع‌ترین بلوک‌چین‌ها که از Rust برای توسعه قراردادهای هو
* **Near Protocol** : بلاك‌چین لایه اول م

### **جمع بندی**

زنگ به قابلیت‌های بی‌نظیر خود به یکی از زبان‌های اصلی برای توسعه زیرساخت‌های بلاک‌چین و قراردادهای هوشمند تبدیل شده است. ایمنی، کارایی، و پذیرفتنی این زبان به توسعه دهندگان کمک می‌شود تا سیستم‌های غیرمتمرکز ایمن، امن‌پذیر، و

سوال24:

زبان برنامه‌نویسی **Rust** به دلیل ویژگی‌های ایمنی حافظه، کارایی بالا، و مدیریت منابع کارآمد

### **ویژگی های Rust مناسب برای هوش مصنوعی**

1. **کارایی بالا و بدون Garbage Collection** : برای پیاده‌سازی مدل‌های ماشینی که نیاز به سرع دارد
2. **حافظه های ایمنی** : از مشکلات حافظه های رایج که در زبان های C و
3. **همزمانی ایمان** : برای پیاده‌سازی الگوریتم‌های موازی و توtokioو async.

### **کاربردهای Rust در هوش مصنوعی**

#### **پیاده‌سازی الگوریتم‌های طراحی ماشین**

در حال توسعه هستند که امکان استفاده از کتابخانه ماشین را میسر می کند

* **Linfa** : مجموعه ای از الگوریتم های ماشین به سبک scikit-learn.
* **RustNN** : یک کتابخانه سا

**نمونه کد با Linfa برای طبقه‌بندی K-means** :

زنگ زدگی

کد را کپی کنید

use linfa::prelude::\*;

use linfa\_clustering::KMeans;

fn main() {

let dataset = Dataset::from((vec![1.0, 2.0, 3.0], vec![4.0, 5.0, 6.0]));

let model = KMeans::params(3).fit(&dataset).unwrap();

println!("{:?}", model.centroids());

}

#### 2. **تحلیل داده‌ها و مفاهیم عمیق**

* **tch-rs** : ارتباطی برای استفاده از **PyTorch** در Rust که امکان ساخت و آموزش شبکه های عص را فراهم می کند.

**نمونه کد استفاده از tch-rs برای یک مدل ساده:**

زنگ زدگی

کد را کپی کنید

use tch::nn::{self, Module, OptimizerConfig};

use tch::{Tensor, Kind, Device};

fn main() {

let vs = nn::VarStore::new(Device::Cpu);

let net = nn::seq().add(nn::linear(vs.root(), 4, 1, Default::default()));

let inputs = Tensor::of\_slice(&[1.0, 2.0, 3.0, 4.0]).view([1, 4]);

let output = net.forward(&inputs);

println!("Output: {:?}", output);

}

#### 3. **سیستم های تولید و استنتاج بلادرنگ**

Rust به دلیل کارایی بالا در برنامه‌هایی که نیاز به استنتاج بلادرنگ دارند، مانند روباتیک یا سیستم‌های تشخیص چهره هستند.

### **ادغام با زبانهای دیگر**

Rust می‌تواند به عنوان یک جزء کار در کنار پایتون برای محاسبات سنگین و بهینه‌سازی عملکرد استفاده شود.

* **PyO3** و **Rust-Python bindings** امکان نوشتن کتابخانه‌های Rust برای ا

### **جمع بندی**

Rust با وجود جامعه کوچک‌تری نسبت به Python، به‌سرعت در حال رشد در حوزه مصنوعی مصنوعی است. ویژگی‌های ایمنی و سرعت بالای آن باعث می‌شود تا یک ایده‌آل برای پروژه‌های بزرگ و تولیدی تبدیل شود، به ویژه در مواردی که به کارایی بالا و مدیریت نیاز دارید

سوال25:

زبان **Rust** ابزارها و چ

### **ابزارهای اصلی تست در Rust**

#### 1. **ماژول تست داخلی Rust**

Rust به صورت پیش‌فرض از ماژول تست داخلی ( #[test]) پشتیبانی می‌کند.

* **ویژگی ها** :
  + ساده و قدرتمند برای نوشتن تست‌های واحدی.
  + پشتیبانی از ویژگی‌های مانند #[should\_panic]برای پایداری پایدار.

**نمونه کد تست داخلی:**

زنگ زدگی

کد را کپی کنید

#[cfg(test)]

mod tests {

#[test]

fn it\_adds\_two() {

assert\_eq!(2 + 2, 4);

}

}

* **مزایا** :
  + در خود زبان Rust تعبیه شده است.
  + نیاز به نصب ابزار خارجی ندارد.
* **مناسب برای** : تستهای واحدی و ساده.

#### 2. **تست های یکپارچه سازی**

تست‌هایی که خارج از ماژول اصلی قرار می‌گیرند و رفتار کل برنامه یا کتابخانه را می‌گیرند.

* **ساختار** : فایل های تست در پوشه های به نام tests/قرار میگیرند.

**نمونه تست یکپارچه سازی** :

زنگ زدگی

کد را کپی کنید

// tests/integration\_test.rs

use my\_crate; // استفاده از کرت اصلی

#[test]

fn it\_works() {

assert!(my\_crate::some\_function());

}

* **مناسب برای** : بررسی میان ماژول‌ها و توابع.

#### 3. **معیار**

یک کتابخانه قدرتمند برای بنچمارکینگ.

* **ویژگی ها** : اندازه گیری عملکرد و بهینه سازی کدها.

**کد نمونه با معیار** :

زنگ زدگی

کد را کپی کنید

use criterion::{black\_box, Criterion, criterion\_group, criterion\_main};

fn my\_benchmark(c: &mut Criterion) {

c.bench\_function("fib 20", |b| b.iter(|| black\_box(20)));

}

criterion\_group!(benches, my\_benchmark);

criterion\_main!(benches);

* **مناسب برای** : ارزیابی نرخ و کارایی کدها.

#### 4. **موکل**

کتابخانه‌ای برای ایجاد Mock

* **ویژگی ها** : ایجاد نسخه های شبیه سازی شده از توابع و اشیاء برا

**نمونه کد مدل Mock** :

زنگ زدگی

کد را کپی کنید

use mockall::predicate::\*;

use mockall::\*;

#[automock]

pub trait MyTrait {

fn get\_data(&self) -> i32;

}

#[test]

fn test\_with\_mock() {

let mut mock = MockMyTrait::new();

mock.expect\_get\_data().return\_const(42);

assert\_eq!(mock.get\_data(), 42);

}

* **مناسب برای** : تست وابستگی های خارجی و شبیه سازی رفتارها.

### **تفاوت های اصلی**

| ابزار | نوع تست | ویژگی ها | کاربرد |
| --- | --- | --- | --- |
| **#[تست]** | واحدی | ساده و داخلی | برای تست واحدهای کوچک کد |
| **تست های یکپارچه سازی** | ادغام | مستقل از ماژول‌ها | تست اجزای مختلف |
| **معیار** | بنچمارکینگ | تحلیل عملکرد | بهینه سازی سرعت |
| **Mockall** | واحدی | شبیه سازی رفتارها | تست وابستگی های خارجی |

### **جمع بندی**

Rust با ابزارهای متنوع برای تست و بنچمارکینگ، توسعه کدهای ایمن و کارآمد را می‌کند. انتخاب ابزار مناسب برای نوع تست و نیازهای پروژه، اما شروع با امکانات داخلی R

سوال26:

زبان برنامه‌نویسی **Rust** برای توسعه وب ابزارها و فریم‌ورک‌های متن

**1. Actix Web**

یک

* **ویژگی ها** :
  + تاسیس بر Actor model.
  + پشتیبانی قوی از WebSocket، HTTP/2 و میان افزار.
  + قابلیت های همزمانی پیشرفته

**کد نمونه:**

زنگ زدگی

کد را کپی کنید

use actix\_web::{web, App, HttpServer, Responder};

async fn greet() -> impl Responder {

"Hello, World!"

}

#[actix\_web::main]

async fn main() -> std::io::Result<()> {

HttpServer::new(|| App::new().route("/", web::get().to(greet)))

.bind("127.0.0.1:8080")?

.run()

.await

}

* **مزایا** : کارایی بسیار بالا و پشتیبانی ا
* **معایب** : پیچیدگی بیشت

**2. موشک**

یک فریم‌ورک قدرتمند و با استفاده از بر سهولت.

* **ویژگی ها** :
  + پشتیبانی از نو
  + مناسب برای ساخت سریع پروتوتایپ‌ها.

**کد نمونه:**

زنگ زدگی

کد را کپی کنید

#[macro\_use] extern crate rocket;

#[get("/")]

fn index() -> &'static str {

"Hello, Rocket!"

}

#[launch]

fn rocket() -> \_ {

rocket::build().mount("/", routes![index])

}

* **مزایا** : خوانایی و کم بالا.
* **معایب** : نسخه پایدار هنوز از برخی ویژگی‌های جدید Rust استفاده نمی‌کند.

**3. تار**

فریم‌ورکی مدرن با استفاده از نوع ایمنی (نوع ایمنی).

* **ویژگی ها** :
  + پشتیبانی از async/wait.
  + قابلیت تعریف مسیرها به صورت ترکیبی و قابل قبول.

**کد نمونه:**

زنگ زدگی

کد را کپی کنید

use warp::Filter;

#[tokio::main]

async fn main() {

let hello = warp::path!("hello" / String).map(|name| format!("Hello, {}!", name));

warp::

warp:

serve(hello).run(([127, 0, 0, 1], 3030)).await;

}

* **مزایا** : ایمنی نوع قوی و طراحی کاربردی.
* **معایب** : مم

**4. Axum**

کتابخانه‌ای بر پایه **توکیو** با مطالعه بر اساس و کارایی.

* **ویژگی ها** :
  + استفاده از async/await
  + طراحی ساده و سازگاری خوب با سایر ابزارهای اکوسیستم Rust.

**کد نمونه:**

زنگ زدگی

کد را کپی کنید

use axum::{handler::get, Router};

u

use std::net::SocketAddr;

#[tokio::main]

async fn main() {

let app = Router::new().route("/", get(|| async { "Hello, Axum!" }));

let addr = SocketAddr::from(([127, 0, 0, 1], 3000));

axum::Server::

axum::Serv

bind(&addr)

.serve(app.into\_make\_service())

.

await

.unwrap();

}

* **مزایا** : ساده، سریع و سازگار با Async و
* **معایب** : جامعه کاربری کوچکتر نسبت به Actix و Rocket.

**مقایسه ابزارها**

| **ابزار** | **ویژگی ها** | **مزایا** | **معایب** |
| --- | --- | --- | --- |
| **اکتیکس وب** | تاسیس بر Actor Model | بسیار سریع و قابل قبول | پیچ |
| **موشک** | ساده و قوی | مناسب برای پروتوتایپ‌سازی س | عدم پشتیبانی از نسخه پایدار async |
| **پیچ و تاب** | ایمنی نوع قوی | طراحی طراحی‌پذیر و مدرن | نیاز به درک مفاهیم تر |
| **اکسوم** | بر پایه توکیو | سخت و سازگاری بالا | نسبتاً جدید |

**جمع بندی**

هر یک از این ابزارها برای نیازهای مناسب هستند. **Actix Web** برای برنامه های**موشک** برای توس**تار** ب**Axum** برای پروژه‌

سوال27: در زبان **Rust**، مکانیسم‌های ر**خطاهای قابل بازیابی** و **خطاهای غیرقابل بازیابی**.

**خطاهای قابل بازیابی (Recoverable Errors)**

این خطاها معمولاً با استفاده از نوع Result<T, E> مدیریت می‌شوند. این نوع، یک مقدار یا خطا را بازمی‌گرداند.

#### **ساختار Result**

rust

Copy code

enum Result<T, E> {

Ok(T),

Err(E),

}

**مثال استفاده از** Result **برای خواندن یک فایل:**

rust

Copy code

use std::fs::File;

use std::io::ErrorKind;

fn main() {

let file\_path = "example.txt";

match File::open(file\_path) {

Ok(file) => println!("File opened successfully: {:?}", file),

Err(error) => match error.kind() {

ErrorKind::NotFound => println!("File not found."),

\_ => println!("An error occurred: {:?}", error),

},

}

}

#### **استفاده از** unwrap **و** expect

برای ساده‌سازی مدیریت خطاها، از متدهای زیر می‌توان استفاده کرد:

* unwrap(): اگر Result مقدار Ok باشد، مقدار را بازمی‌گرداند، در غیر این صورت برنامه را متوقف می‌کند.
* expect("پیام خطا"): مشابه unwrap اما پیام خطای سفارشی نمایش می‌دهد.

**مثال:**

rust

Copy code

let file = File::open("example.txt").expect("Failed to open file");

### **2. خطاهای غیرقابل بازیابی (Unrecoverable Errors)**

برای این نوع خطاها از panic! استفاده می‌شود که باعث متوقف شدن برنامه می‌شود.

**مثال استفاده از** panic!:

rust

Copy code

fn main() {

let numbers = vec![1, 2, 3];

println!("Number: {}", numbers[5]); // panic: دسترسی به ایندکس نامعتبر

}

### **کد تمیزتر با** ?

عملگر ? یک روش مختصر برای مدیریت خطاهای Result است.

**مثال استفاده از** ?**:**

rust

Copy code

use std::fs::File;

use std::io::{self, Read};

fn read\_file\_content(path: &str) -> Result<String, io::Error> {

let mut file = File::open(path)?;

let mut content = String::new();

file.read\_to\_string(&mut content)?;

Ok(content)

}

fn main() {

match read\_file\_content("example.txt") {

Ok(content) => println!("File content: {}", content),

Err(e) => println!("Error reading file: {:?}", e),

}

}

### **جمع‌بندی**

* Result<T, E> برای خطاهای قابل بازیابی مناسب است.
* panic! برای شرایطی که ادامه اجرای برنامه منطقی نیست.
* **عملگر** ? کد را ساده‌تر می‌کند و مدیریت خطا را کارآمدتر می‌سازد

سوال28:

زبان Rust با وجود اینکه یک زبان چند‌الگویی (Multi-Paradigm) است، قابلیت‌های قدرتمندی برای برنامه‌نویسی تابعی (Functional Programming) فراهم می‌کند. این قابلیت‌ها به توسعه‌دهندگان اجازه می‌دهند کدی ایمن‌تر، ساده‌تر، و انعطاف‌پذیرتر بنویسند.

1.توابع مرتبه بالاتر (Higher-Order Functions)

توابعی که می‌توانند به عنوان آرگومان پذیرفته شوند یا تابع دیگری را بازگردانند.

مثال: استفاده از map برای تبدیل عناصر یک بردار

rust

Copy code

fn main() {

let numbers = vec![1, 2, 3, 4];

let squares: Vec<i32> = numbers.iter().map(|x| x \* x).collect();

println!("{:?}", squares); // [1, 4, 9, 16]

}

* map تابعی است که یک closure می‌گیرد و به هر عنصر اعمال می‌کند.

2. Closure (بستن)

توابع بدون نام که می‌توانند محیط بیرونی خود را اسیر (capture) کنند.

مثال Closure:

rust

Copy code

fn main() {

let add = |x: i32, y: i32| x + y;

println!("Sum: {}", add(5, 3)); // Sum: 8

}

* Closure فوق دو عدد را جمع می‌کند.

3. ترکیب و استفاده از Iterators

Rust دارای کتابخانه‌های استاندارد برای کار با آبجکت‌های iterator است که به سبک تابعی طراحی شده‌اند.

مثال استفاده از filter و sum برای محاسبه مجموع اعداد زوج:

rust

Copy code

fn main() {

let numbers = vec![1, 2, 3, 4, 5, 6];

let sum\_of\_evens: i32 = numbers.into\_iter().filter(|&x| x % 2 == 0).sum();

println!("Sum of even numbers: {}", sum\_of\_evens); // Sum of even numbers: 12

}

4. Immutability (تغییرناپذیری)

برنامه‌نویسی تابعی تمایل به استفاده از داده‌های تغییرناپذیر دارد. در Rust، متغیرها به‌طور پیش‌فرض غیرقابل تغییر هستند.

مثال تغییرناپذیری:

rust

Copy code

fn main() {

let x = 5;

// x = 6; // خطا: x به‌طور پیش‌فرض تغییرناپذیر است

}

5. Pattern Matching (الگوسازی)

یکی دیگر از مفاهیم تابعی قوی که در Rust وجود دارد، الگوسازی است که با match و if let انجام می‌شود.

مثال match:

rust

Copy code

fn main() {

let number = Some(10);

match number {

Some(n) if n > 5 => println!("Greater than 5: {}", n),

Some(\_) => println!("Less or equal to 5"),

None => println!("No value"),

}

}

جمع‌بندی

Rust قابلیت‌های بسیاری برای برنامه‌نویسی تابعی دارد، از جمله توابع مرتبه بالا، closureها، immutability، و الگوهای تطبیق. این امکانات به نوشتن کدی قابل اطمینان‌تر، خواناتر، و امن‌تر کمک می‌کنند، در حالی که همچنان بهره‌وری و کارایی زبان‌های سیستمی را حفظ می‌کنند

سوال 29:

بله، **برنامه‌نویسی شی‌گرا** (Object-Oriented Programming) در زبان **Rust** امکان‌پذیر است، هرچند که این زبان رویکرد خاص خود را دارد و مستقیماً از برخی از مفاهیم سنتی شی‌گرایی مانند **کلاس‌ها** پشتیبانی نمی‌کند. به جای آن، Rust از ساختارهای قدرتمندتری مانند **ساختارها (Structs)** و **ویژگی‌ها (Traits)** برای پیاده‌سازی مفاهیم اصلی شی‌گرایی بهره می‌برد.

### **1. مفاهیم اصلی شی‌گرایی در Rust**

#### **الف. کپسوله‌سازی (Encapsulation)**

در Rust، می‌توان از ساختارها (Structs) برای تعریف اشیاء استفاده کرد و دسترسی به داده‌ها را با استفاده از قابلیت سطح دسترسی (public/private) کنترل کرد.

**مثال کپسوله‌سازی با** struct**:**

rust

Copy code

struct Rectangle {

width: u32,

height: u32,

}

impl Rectangle {

// متدهای مرتبط با Rectangle

fn area(&self) -> u32 {

self.width \* self.height

}

fn set\_width(&mut self, width: u32) {

self.width = width;

}

}

fn main() {

let mut rect = Rectangle { width: 10, height: 20 };

println!("Area: {}", rect.area());

rect.set\_width(15);

println!("Updated Area: {}", rect.area());

}

* **کپسوله‌سازی**: فیلدهای Rectangle می‌توانند به صورت خصوصی تعریف شوند.
* **متدها**: با استفاده از impl برای اشیاء تعریف می‌شوند.

#### **ب. وراثت (Inheritance)**

Rust از وراثت به شیوه سنتی پشتیبانی نمی‌کند اما از طریق **ویژگی‌ها (Traits)** به پیاده‌سازی **چندریختی (Polymorphism)** می‌پردازد. Traits مشابه **اینترفیس‌ها** در زبان‌های دیگر عمل می‌کنند.

**مثال تعریف Trait:**

rust

Copy code

trait Shape {

fn area(&self) -> f64;

}

struct Circle {

radius: f64,

}

struct Square {

side: f64,

}

impl Shape for Circle {

fn area(&self) -> f64 {

3.14 \* self.radius \* self.radius

}

}

impl Shape for Square {

fn area(&self) -> f64 {

self.side \* self.side

}

}

fn print\_area(shape: &impl Shape) {

println!("Area: {}", shape.area());

}

fn main() {

let circle = Circle { radius: 5.0 };

let square = Square { side: 4.0 };

print\_area(&circle);

print\_area(&square);

}

* **Traits** جایگزینی برای وراثت هستند و به تعریف رفتارهای مشترک کمک می‌کنند.

#### **ج. چندریختی (Polymorphism)**

Rust از چندریختی ایستا (Static Polymorphism) با **ژنریک‌ها** و چندریختی پویا (Dynamic Polymorphism) با **ویژگی‌های شی‌وار (Trait Objects)** پشتیبانی می‌کند.

**چندریختی پویا با Trait Objects:**

rust

Copy code

fn print\_area\_dyn(shape: &dyn Shape) {

println!("Area: {}", shape.area());

}

### **جمع‌بندی**

Rust یک زبان چندالگویی است که به شی‌گرایی با استفاده از **structs**، **traits** و **impl** نزدیک می‌شود. در عین حال، با حذف وراثت کلاسیک و استفاده از **ترکیب (Composition)** و **ویژگی‌ها**، از پیچیدگی‌ها و مشکلات متداول وراثت سنتی جلوگیری می‌کند. این روش‌ها امنیت و کارایی را در کنار انعطاف‌پذیری به ارمغان می‌آورند

سوال30:

زبان Rust با ارائه کارایی بالا و ایمنی حافظه، برای کاربردهای یادگیری ماشین (Machine Learning) و علم داده در حال محبوب شدن است. چندین کتابخانه به‌طور فعال برای این حوزه توسعه یافته‌اند.

کتابخانه‌های محبوب یادگیری ماشین در Rust

1. Linfa: چارچوبی برای الگوریتم‌های کلاسیک یادگیری ماشین.
2. Smart Core: مجموعه‌ای از الگوریتم‌های یادگیری ماشین با تمرکز بر کارایی.
3. Rust Learn: ابزارهای استاندارد برای یادگیری ماشین.
4. Tch -rs: رابط Rust برای کتابخانه Py Torch.
5. ndarray: برای محاسبات آرایه‌های چندبعدی، کاربردی در عملیات ریاضی و پردازش داده‌ها.

مثال ساده: استفاده از Linfa برای رگرسیون خطی

rust

Copy code

use linfa::traits::Fit;

use linfa\_linear::LinearRegression;

use ndarray::array;

fn main() {

// داده‌های آموزشی (دو ویژگی و مقادیر هدف)

let x = array![[1.0], [2.0], [3.0], [4.0]];

let y = array![2.0, 4.0, 6.0, 8.0];

// ایجاد مدل رگرسیون خطی

let model = LinearRegression::default().fit(&x, &y).unwrap();

// پیش‌بینی برای داده‌های جدید

let x\_new = array![[5.0]];

let prediction = model.predict(&x\_new);

println!("Prediction for input 5.0: {:?}", prediction);

}

توضیحات

* کتابخانه ndarray برای مدیریت ماتریس‌ها و بردارهای چندبعدی به کار می‌رود.
* Linfa برای مدل‌های یادگیری ماشین استفاده می‌شود، و این مثال ساده‌ای از رگرسیون خطی برای پیش‌بینی مقادیر است.

Rust با توجه به ابزارهای رو به رشد، به‌تدریج در حوزه یادگیری ماشین جایگاه خود را تثبیت می‌کند و برای کاربردهای بهینه‌سازی، پردازش موازی و سیستم‌های با کارایی بالا گزینه مناسبی است.

کتابخانه‌های معروف یادگیری ماشین در Rust

1. Linfa: چارچوبی برای الگوریتم‌های یادگیری ماشین کلاسیک.
2. SmartCore: مجموعه‌ای جامع از الگوریتم‌های یادگیری ماشین.
3. Tch-rs: رابط Rust برای PyTorch برای یادگیری عمیق.
4. RustLearn: ابزارهای ساده و سریع برای یادگیری ماشین.
5. ndarray: برای محاسبات ماتریسی و عملیات برداری.
6. Polars: برای پردازش داده‌ها و ساختارهای جدولی (DataFrame).

مثال ساده: استفاده از linfa برای رگرسیون خطی

rust

Copy code

use linfa::traits::{Fit, Predict};

use linfa\_linear::LinearRegression;

use ndarray::array;

fn main() {

// داده‌های آموزشی: ویژگی‌ها و برچسب‌ها

let x\_train = array![[1.0], [2.0], [3.0], [4.0]];

let y\_train = array![1.5, 3.0, 4.5, 6.0];

// ساخت مدل رگرسیون خطی

let model = LinearRegression::default().fit(&x\_train, &y\_train).unwrap();

// پیش‌بینی مقدار برای ورودی جدید

let x\_new = array![[5.0]];

let y\_pred = model.predict(&x\_new);

println!("پیش‌بینی برای ورودی 5.0: {:?}", y\_pred);

}

توضیحات

* ndarray برای آرایه‌ها و عملیات ماتریسی.
* Linfa برای الگوریتم‌های یادگیری ماشین با مدل‌های متنوع.
* مدل رگرسیون خطی روی داده‌های آموزشی ساده آموزش دیده و برای پیش‌بینی استفاده می‌شود.

این مثال نمونه‌ای از رویکرد تمیز و ایمن Rust برای کارهای یادگیری ماشین را نشان می‌دهد.

سوال31:

زبان Rust به دلیل تمرکز بر ایمنی و مدیریت دقیق حافظه، ابزارهای قدرتمندی برای برنامه‌نویسی چند‌ریسمانی (Multi-Threading) فراهم می‌کند. این زبان با استفاده از مدل مالکیت و سیستم نوع خود از شرایط رقابتی (Race Conditions) و مشکلات معمول چندریسمانی جلوگیری می‌کند.

مثال ساده از Multi-Threading در Rust

rust

Copy code

use std::thread;

use std::time::Duration;

fn main() {

// ایجاد یک رشته (thread) جدید

let handle = thread::spawn(|| {

for i in 1..5 {

println!("پیام از رشته جدید: {}", i);

thread::sleep(Duration::from\_millis(500));

}

});

// اجرای کد در رشته اصلی

for i in 1..5 {

println!("پیام از رشته اصلی: {}", i);

thread::sleep(Duration::from\_millis(1000));

}

// منتظر ماندن برای اتمام اجرای رشته جدید

handle.join().unwrap();

}

توضیح کد

1. ایجاد رشته جدید با thread::spawn: این تابع یک کلوسر می‌پذیرد و کدی را که باید در رشته جدید اجرا شود تعریف می‌کند.
2. join: اطمینان حاصل می‌کند که رشته فرعی قبل از پایان برنامه اجرا شود.
3. sleep: برای ایجاد تأخیر و مشاهده خروجی همزمان از رشته‌ها.

نمونه خروجی

python

Copy code

پیام از رشته اصلی: 1

پیام از رشته جدید: 1

پیام از رشته اصلی: 2

پیام از رشته جدید: 2

پیام از رشته اصلی: 3

پیام از رشته جدید: 3

...

مدیریت ایمنی حافظه در Multi-Threading

Rust با استفاده از مدل مالکیت و ویژگی‌های borrow-checker مانع از دسترسی ناسازگار به داده‌های اشتراکی می‌شود. به عنوان مثال، برای اشتراک داده میان چند رشته، باید از Arc<Mutex<T>> استفاده کرد:

* Arc برای اشتراک‌گذاری داده‌ها به‌طور امن میان رشته‌ها.
* Mutex برای اطمینان از دسترسی متقارن به داده‌های مشترک.

جمع‌بندی

Rust رویکردی ایمن و بهینه برای چندریسمانی فراهم می‌کند و از مشکلات معمول زبان‌های دیگر در این حوزه جلوگیری می‌کند، که آن را برای توسعه سیستم‌های همزمان و با کارایی بالا ایده‌آل می‌کند.

32سوال: **برنامه‌نویسی موازی** (Parallel Programming) در **Rust** به معنای اجرای چند وظیفه به طور همزمان برای بهره‌وری بیشتر از منابع پردازشی است. Rust با ارائه ابزارهایی مانند rayon مدیریت ایمن و کارآمد کد موازی را ممکن می‌سازد.

### **کتابخانه** rayon **برای برنامه‌نویسی موازی**

کتابخانه‌ی rayon برای اجرای موازی آسان و ایمن روی داده‌ها استفاده می‌شود.

#### **مثال ساده: محاسبه جمع موازی اعداد**

rust

Copy code

use rayon::prelude::\*;

fn main() {

let numbers: Vec<i32> = (1..=1\_000\_000).collect();

// محاسبه مجموع به صورت موازی

let sum: i32 = numbers.par\_iter().sum();

println!("مجموع اعداد از 1 تا 1,000,000: {}", sum);

}

### **توضیحات کد**

1. rayon::prelude::\*: برای استفاده از ویژگی‌های موازی.
2. par\_iter(): یک **iterator موازی** برای پردازش عناصر بردار به‌طور همزمان.
3. sum(): جمع موازی مقادیر موجود در بردار.

### **چرا** rayon**؟**

* این کتابخانه کدهای دنباله‌ای (sequential) را به صورت موازی مدیریت می‌کند بدون نیاز به تغییرات بزرگ.
* **ایمنی حافظه** را در عملیات موازی تضمین می‌کند، برخلاف بسیاری از زبان‌های دیگر که ممکن است به خطاهای رقابتی منجر شوند.

### **مزایای برنامه‌نویسی موازی با Rust**

* **ایمنی در برابر شرایط رقابتی**: مالکیت داده‌ها در Rust تضمین می‌کند که داده‌های اشتراکی به‌طور ایمن استفاده شوند.
* **کارایی بالا**: Rust در سطح سیستمی بهینه است و عملکرد مناسبی در برنامه‌های موازی ارائه می‌دهد.

### **خلاصه**

Rust با ابزارهایی مانند rayon، برنامه‌نویسی موازی را به روشی ایمن و کارآمد تبدیل کرده است که برای کار با داده‌های بزرگ و وظایف موازی بسیار مناسب است

سوال33:

**Meta Programming** در زبان **Rust** به تولید یا تغییر کد در زمان کامپایل و همچنین به ابزارهای بررسی و بهینه‌سازی عملکرد (Profiling) اشاره دارد. برخلاف زبان‌هایی مانند Python یا Java که از **Reflection** برای بررسی و تغییر ساختار برنامه در زمان اجرا استفاده می‌کنند، Rust به دلیل ماهیت کامپایل-محور بودن و تاکید بر کارایی، فاقد reflection پویا است. اما ویژگی‌های دیگری مانند **ماکروها** و **ویژگی‌ها (attributes)** برای meta-programming ارائه می‌دهد.

### **1. ماکروهای Procedural در Rust (Meta Programming)**

Rust از ماکروهای پیچیده‌ای برای تولید کد استفاده می‌کند که قدرتمند و ایمن هستند.

#### **مثال: استفاده از ماکرو برای تولید کد**

rust

Copy code

#[derive(Debug)]

struct Point {

x: i32,

y: i32,

}

fn main() {

let point = Point { x: 10, y: 20 };

println!("{:?}", point); // از ماکرو برای تولید کد نمایش استفاده شده است

}

در اینجا:

* #[derive(Debug)] ماکرویی است که کد برای نمایش ساختار Point تولید می‌کند.
* این نوع ماکروها به جای reflection در زمان اجرا، در زمان کامپایل کد تولید می‌کنند.

### **2. ابزارهای پروفایلینگ (Profiling) در Rust**

پروفایلینگ به بررسی عملکرد کد برای شناسایی بخش‌های کند و ناکارآمد کمک می‌کند.

#### **ابزارهای پروفایلینگ محبوب**

* perf: برای سیستم‌های لینوکس، همراه با cargo استفاده می‌شود.
* flamegraph: برای تولید نمای گرافیکی از داده‌های پروفایل.
* cargo-profiler: ابزاری ساده برای پروفایلینگ در Rust.

#### **پروفایل ساده با استفاده از** flamegraph

bash

Copy code

cargo install flamegraph

cargo flamegraph

این دستورها به شما یک نمای گرافیکی از زمان مصرف‌شده در بخش‌های مختلف برنامه ارائه می‌دهند.

### **مقایسه Reflection و Meta Programming در Rust**

| ویژگی | Rust | زبان‌های پویا (مانند Python) |
| --- | --- | --- |
| Reflection | در زمان کامپایل، با استفاده از ماکروها | در زمان اجرا، با introspection |
| Safety | بسیار ایمن، بدون هزینه زمان اجرا | هزینه بالا برای انعطاف بیشتر |
| Profiling Tools | ابزارهای متعدد مانند perf و flamegraph | ابزارهای داخلی و کتابخانه‌های متنوع |

### **خلاصه**

Rust رویکردی متفاوت و ایمن به meta-programming دارد، که با تولید کد در زمان کامپایل از ویژگی‌های reflection سنتی فاصله می‌گیرد. این امر آن را برای سیستم‌های کارایی‌محور و ایمن مناسب می‌سازد

سوال34:

**Rust** با معرفی مدل **مالکیت (Ownership)**، **قرض گرفتن (Borrowing)**، و **سیستم بررسی‌کننده وام (Borrow Checker)** راه‌حلی کارآمد برای مشکلات ایمنی حافظه ارائه می‌دهد. این زبان بدون نیاز به استفاده از جمع‌آوری زباله (Garbage Collection) ایمنی حافظه را تضمین می‌کند و از مشکلاتی مانند **شرایط رقابتی (Race Conditions)**، **دسترسی به داده‌های آزاد شده (Dangling Pointers)** و **نشت حافظه (Memory Leaks)** جلوگیری می‌کند.

### **مفاهیم کلیدی ایمنی حافظه در Rust**

1. **مالکیت (Ownership)**: هر مقدار دقیقاً یک مالک دارد.
2. **قرض گرفتن (Borrowing)**: می‌توان به جای انتقال مالکیت، به مقدار قرض گرفت.
3. **قرض گرفتن متغیر یا غیرمتغیر**: یک مقدار می‌تواند چندین وام غیرمتغیر یا فقط یک وام متغیر داشته باشد.

### **مثال ساده برای ایمنی حافظه در Rust**

#### کدی که از انتقال مالکیت استفاده می‌کند

rust

Copy code

fn main() {

let s1 = String::from("Hello, Rust!"); // مالکیت رشته

let s2 = s1; // انتقال مالکیت به s2

// println!("{}", s1); // خطا: s1 دیگر مالک رشته نیست و دسترسی به آن غیرمجاز است

println!("{}", s2); // دسترسی به s2 مجاز است

}

**توضیح:**

* مالکیت رشته s1 به s2 منتقل شده است. بنابراین، دسترسی به s1 پس از انتقال غیرمجاز است. این از دسترسی به داده‌های نامعتبر جلوگیری می‌کند.

#### قرض گرفتن ایمن (Borrowing)

rust

Copy code

fn print\_length(s: &String) {

println!("طول رشته: {}", s.len());

}

fn main() {

let s = String::from("Hello, Rust!");

print\_length(&s); // قرض گرفتن غیرمتغیر برای چاپ طول رشته

println!("رشته اصلی: {}", s); // s هنوز در دسترس است

}

**توضیح:**

* متغیر s به صورت وام غیرمتغیر به تابع print\_length داده شده است. این وام بدون انتقال مالکیت انجام می‌شود و پس از بازگشت تابع، همچنان می‌توان به s دسترسی داشت.

### **مدیریت داده‌های اشتراکی در چندریسمانی**

برای دسترسی همزمان به داده‌های مشترک، Rust از ترکیب Arc و Mutex استفاده می‌کند، که از شرایط رقابتی جلوگیری می‌کند.

### **نتیجه‌گیری**

مدل مالکیت و سیستم نوع Rust تضمین می‌کند که برنامه‌های نوشته‌شده با این زبان ایمن از نظر حافظه و بدون هزینه اضافی برای جمع‌آوری زباله هستند. این مدل از مشکلات رایج در زبان‌های دیگر جلوگیری می‌کند و Rust را برای برنامه‌های سیستمی و کارایی‌محور مناسب می‌سازد.

زبان **Rust** با استفاده از مفاهیمی مانند **مالکیت (Ownership)**، **قرض گرفتن (Borrowing)** و **بررسی‌کننده وام (Borrow Checker)** راه‌حل‌های موثری برای جلوگیری از مشکلات ایمنی حافظه مانند **نشانگرهای تهی (Null Pointers)**، **نشت حافظه (Memory Leak)**، **استفاده پس از آزادسازی (Use-After-Free)**، **مسابقه داده (Data Race)** و **آزادسازی دوگانه (Double Free)** ارائه می‌دهد.

در ادامه این مشکلات به همراه راه‌حل Rust و مثال‌های مرتبط توضیح داده شده‌اند:

### **1. نشانگر تهی (Null Pointers)**

در Rust، به جای استفاده از نشانگرهای تهی، از نوع Option<T> برای مدیریت مقادیر اختیاری استفاده می‌شود.

#### **مثال نشانگر تهی ایمن با** Option

rust

Copy code

fn safe\_null\_pointer() {

let value: Option<i32> = Some(10); // مقدار اختیاری

if let Some(num) = value {

println!("مقدار: {}", num);

} else {

println!("بدون مقدار!");

}

}

**توضیح**: Option<T> مانع از دسترسی به اشاره‌گرهای تهی می‌شود.

### **2. نشت حافظه**

Rust به‌طور خودکار منابع را با استفاده از مدل مالکیت آزاد می‌کند. با این حال، استفاده از Rc یا Arc بدون مدیریت صحیح می‌تواند به نشت حافظه منجر شود. این زبان از **بدون هزینه آزادسازی دوگانه** اطمینان حاصل می‌کند.

#### **راه‌حل با** Drop

rust

Copy code

struct Resource;

impl Drop for Resource {

fn drop(&mut self) {

println!("منبع آزاد شد.");

}

}

fn main() {

let \_res = Resource;

println!("برنامه اجرا شد.");

}

**توضیح**: Drop مدیریت خودکار حافظه را انجام می‌دهد.

### **3. استفاده پس از آزادسازی (Use-After-Free)**

Rust به دلیل مدل مالکیت، اجازه نمی‌دهد به داده‌هایی که مالکیتشان از بین رفته است، دسترسی داشته باشید.

#### **مثال استفاده پس از آزادسازی (Use-After-Free)**

rust

Copy code

fn main() {

let s1 = String::from("سلام");

let s2 = s1; // انتقال مالکیت به s2

// println!("{}", s1); // خطا: s1 دیگر مالک نیست

println!("{}", s2); // فقط s2 قابل استفاده است

}

### **4. مسابقه داده (Data Race)**

Rust با ترکیب Mutex و Arc از شرایط مسابقه جلوگیری می‌کند.

rust

Copy code

use std::sync::{Arc, Mutex};

use std::thread;

fn main() {

let counter = Arc::new(Mutex::new(0));

let mut handles = vec![];

for \_ in 0..10 {

let counter = Arc::clone(&counter);

let handle = thread::spawn(move || {

let mut num = counter.lock().unwrap();

\*num += 1;

});

handles.push(handle);

}

for handle in handles {

handle.join().unwrap();

}

println!("نتیجه نهایی: {}", \*counter.lock().unwrap());

}

**توضیح**: استفاده از Arc<Mutex<T>> اجازه دسترسی ایمن به داده‌های مشترک را می‌دهد.

### **5. آزادسازی دوگانه (Double Free)**

Rust به‌طور خودکار از آزادسازی دوگانه جلوگیری می‌کند زیرا فقط یک مالک برای هر منبع وجود دارد.

### **نتیجه‌گیری**

مدل مالکیت Rust همراه با سیستم نوع ایمنی حافظه را تضمین می‌کند و از مشکلات رایج در زبان‌های دیگر جلوگیری می‌کند. این رویکرد باعث می‌شود برنامه‌های Rust ایمن و کارآمد باشند.

**سوال35:**

در ادامه عملکرد کتابخانه‌های ذکر شده در زبان **Rust** با یک مثال ساده برای هر کدام توضیح داده می‌شود:

### ****1. Geo****

عملیات جغرافیایی مانند محاسبه فاصله‌ها.

**مثال:**

rust

Copy code

use geo::{point, HaversineDistance};

fn main() {

let p1 = point!(x: 34.0, y: -118.0);

let p2 = point!(x: 40.0, y: -73.0);

let distance = p1.haversine\_distance(&p2); // فاصله بر حسب متر

println!("فاصله بین نقاط: {} متر", distance);

}

### ****2. Digest****

هش کردن داده‌ها.

**مثال:**

rust

Copy code

use sha2::{Sha256, Digest};

fn main() {

let mut hasher = Sha256::new();

hasher.update(b"Hello, Rust!");

let result = hasher.finalize();

println!("SHA-256: {:x}", result);

}

### ****3. Time****

کار با تاریخ و زمان.

**مثال:**

rust

Copy code

use time::OffsetDateTime;

fn main() {

let now = OffsetDateTime::now\_utc();

println!("زمان فعلی UTC: {}", now);

}

### ****4. Serde****

تبدیل داده‌ها بین فرمت‌های مختلف.

**مثال:**

rust

Copy code

use serde::{Serialize, Deserialize};

#[derive(Serialize, Deserialize)]

struct Person {

name: String,

age: u8,

}

fn main() {

let data = r#"{"name": "Alice", "age": 30}"#;

let person: Person = serde\_json::from\_str(data).unwrap();

println!("نام: {}, سن: {}", person.name, person.age);

}

### ****5. Hound****

مدیریت فایل‌های صوتی WAV.

**مثال:**

rust

Copy code

use hound::WavWriter;

fn main() {

let spec = hound::WavSpec {

channels: 1,

sample\_rate: 44100,

bits\_per\_sample: 16,

};

let mut writer = WavWriter::create("output.wav", spec).unwrap();

writer.write\_sample(1000i16).unwrap();

}

### ****6. Signature****

مدیریت امضای دیجیتال. برای استفاده با **ring** مناسب است.

### ****7. URL****

مدیریت URLها.

**مثال:**

rust

Copy code

use url::Url;

fn main() {

let url = Url::parse("https://www.rust-lang.org/learn").unwrap();

println!("دامنه: {}", url.host\_str().unwrap());

}

### ****8. Lazy\_static****

تعریف متغیرهای ثابت با مقداردهی اولیه در زمان اجرا.

**مثال:**

rust

Copy code

#[macro\_use]

extern crate lazy\_static;

use std::collections::HashMap;

lazy\_static! {

static ref MAP: HashMap<&'static str, i32> = {

let mut m = HashMap::new();

m.insert("key", 42);

m

};

}

fn main() {

println!("مقدار: {}", MAP["key"]);

}

### ****9. Rustls****

کتابخانه TLS برای اتصالات امن.

### ****10. Openssl****

رمزنگاری و توابع امنیتی.

### ****11. Reqwest****

برای درخواست‌های HTTP.

**مثال:**

rust

Copy code

#[tokio::main]

async fn main() {

let body = reqwest::get("https://www.rust-lang.org").await.unwrap().text().await.unwrap();

println!("متن پاسخ: {}", body);

}

### ****12. Thiserror****

مدیریت خطاها با نوع‌های سفارشی.

### ****13. Chrono****

مدیریت زمان و تاریخ.

**مثال:**

rust

Copy code

use chrono::prelude::\*;

fn main() {

let now = Utc::now();

println!("زمان فعلی: {}", now);

}

### ****14. Anyhow****

مدیریت خطاهای عمومی.

### ****15. Itertools****

ابزارهای اضافی برای کار با تکرارگرها.

**مثال:**

rust

Copy code

use itertools::Itertools;

fn main() {

let numbers = vec![1, 2, 3];

println!("ترکیب: {}", numbers.iter().join(", "));

}

این کتابخانه‌ها قابلیت‌های متنوعی از جمله مدیریت داده‌ها، امنیت، پردازش HTTP، و تاریخ و زمان را فراهم می‌کنند که به ساده‌تر کردن برنامه‌نویسی با Rust کمک می‌کنند

**سوال36:**

زبان Rust امکان فراخوانی کدهای نوشته شده در سایر زبان‌های برنامه‌نویسی را از طریق ویژگی‌های FFI (Foreign Function Interface) فراهم می‌کند. این قابلیت به Rust اجازه می‌دهد تا به توابع و کتابخانه‌های خارجی نوشته شده در زبان‌هایی مانند C، C++ و Python دسترسی داشته باشد و از آنها استفاده کند.

مراحل و روش‌های فراخوانی کدهای زبان دیگر در Rust

1. فراخوانی کد C با FFI
   * Rust به صورت مستقیم از کد C پشتیبانی می‌کند. برای این کار، از کلمه کلیدی extern استفاده می‌شود.

مثال: فراخوانی تابع C در Rust فرض کنید تابعی در زبان C داریم:

c

Copy code

// hello.c

#include <stdio.h>

void say\_hello() {

printf("Hello from C!\n");

}

حال می‌توانیم این تابع را در Rust فراخوانی کنیم:

rust

Copy code

extern "C" {

fn say\_hello();

}

fn main() {

unsafe {

say\_hello();

}

}

توضیح:

* کلمه کلیدی extern "C" مشخص می‌کند که این تابع از قرارداد فراخوانی زبان C استفاده می‌کند.
* unsafe نشان‌دهنده فراخوانی کد ناامن (از دید Rust) است، زیرا تضمین‌های ایمنی حافظه توسط Rust بررسی نمی‌شوند.

1. فراخوانی کد C++
   * برای استفاده از کد C++ نیاز به ابزارهای اضافی مانند cxx، bindgen یا cpp crate دارید.

مثال با cxx crate:

rust

Copy code

#[cxx::bridge]

mod ffi {

extern "C++" {

fn hello\_from\_cpp();

}

}

fn main() {

ffi::hello\_from\_cpp();

}

توجه: این مثال نیاز به تنظیمات cxx در فایل‌های C++ دارد.

1. فراخوانی کدهای Python با استفاده از PyO3
   * PyO3 یک کتابخانه Rust است که به شما اجازه می‌دهد کدهای Python را اجرا کنید.

مثال: اجرای کد Python در Rust

rust

Copy code

use pyo3::prelude::\*;

fn main() -> PyResult<()> {

Python::with\_gil(|py| {

let sys = py.import("sys")?;

let version: String = sys.get("version")?.extract()?;

println!("Python version: {}", version);

Ok(())

})

}

1. فراخوانی کدهای زبان‌های دیگر
   * زبان‌های دیگر مانند Java، Go یا حتی WebAssembly نیز با استفاده از ابزارهای مناسب و crateهای مرتبط می‌توانند از طریق Rust تعامل داشته باشند.

جمع‌بندی

قابلیت FFI در Rust باعث انعطاف‌پذیری بالا در ترکیب کدهای Rust با زبان‌های دیگر می‌شود، اما استفاده از آن نیازمند مدیریت دقیق برای جلوگیری از مشکلات ایمنی حافظه است. ابزارهایی مانند bindgen، cxx و PyO3 فرآیند کار با زبان‌های مختلف را ساده می‌کنند.

**سوال37:**

امکان فراخوانی کدهای **Rust** در زبان‌های دیگر مانند **Java**، **C#**، **C** و **Python** با استفاده از ابزارها و تکنیک‌های مختلف فراهم شده است. این روش‌ها از کامپایل کد Rust به کتابخانه‌های پویا یا استاتیک تا استفاده از رابط‌های خارجی برای برقراری ارتباط بین زبان‌ها استفاده می‌کنند.

### 1. **فراخوانی کد Rust در C**

Rust با زبان C سازگاری طبیعی دارد و می‌توان با استفاده از FFI (Foreign Function Interface) کد Rust را در برنامه‌های C فراخوانی کرد.

#### مثال:

کد Rust:

rust

Copy code

#[no\_mangle]

pub extern "C" fn add(a: i32, b: i32) -> i32 {

a + b

}

این کد را به صورت یک کتابخانه پویا کامپایل کنید:

bash

Copy code

rustc --crate-type=cdylib lib.rs -o libmath.so

کد C برای فراخوانی:

c

Copy code

#include <stdio.h>

extern int add(int a, int b);

int main() {

printf("Result: %d\n", add(5, 3));

return 0;

}

### 2. **فراخوانی کد Rust در Python**

با استفاده از **PyO3** می‌توان کد Rust را در Python فراخوانی کرد.

#### مثال:

کد Rust:

rust

Copy code

use pyo3::prelude::\*;

#[pyfunction]

fn double(x: i32) -> i32 {

x \* 2

}

#[pymodule]

fn mymodule(\_py: Python, m: &PyModule) -> PyResult<()> {

m.add\_function(wrap\_pyfunction!(double, m)?)?;

Ok(())

}

برای کامپایل کردن از maturin استفاده کنید و سپس کد Python:

python

Copy code

import mymodule

print(mymodule.double(4)) # Output: 8

### 3. **فراخوانی کد Rust در Java**

برای فراخوانی کد Rust در Java از JNI (Java Native Interface) استفاده می‌شود.

#### مثال:

کد Rust:

rust

Copy code

#[no\_mangle]

pub extern "C" fn hello\_from\_rust() {

println!("Hello from Rust!");

}

کتابخانه پویا تولید کنید و سپس در Java با استفاده از System.loadLibrary فراخوانی کنید:

java

Copy code

public class Main {

static {

System.loadLibrary("hello");

}

public native void hello\_from\_rust();

public static void main(String[] args) {

new Main().hello\_from\_rust();

}

}

### 4. **فراخوانی کد Rust در C#**

برای C#، از کتابخانه‌های پویا یا ابزارهایی مانند UnmanagedExports استفاده می‌شود.

#### مثال:

کد Rust:

rust

Copy code

#[no\_mangle]

pub extern "C" fn greet() {

println!("Hello from Rust!");

}

کد C#:

csharp

Copy code

using System.Runtime.InteropServices;

class Program {

[DllImport("mylibrary.dll")]

public static extern void greet();

static void Main() {

greet();

}

}

### **جمع‌بندی**

* **C**: با استفاده از FFI به‌طور مستقیم از Rust پشتیبانی می‌کند.
* **Python**: از PyO3 یا cffi برای ارتباط استفاده می‌شود.
* **Java**: از JNI برای برقراری ارتباط بین Rust و Java استفاده می‌شود.
* **C#**: از DllImport برای بارگذاری کتابخانه‌های پویا Rust استفاده می‌شود.

این روش‌ها امکان استفاده از قدرت Rust را در زبان‌های مختلف فراهم می‌کنند و در پروژه‌هایی که نیاز به عملکرد بالا و مدیریت ایمنی حافظه دارند، بسیار مفید هستند

**سوال38:**

**Lazy Loading** (بارگذاری تنبل) یک الگوی طراحی است که در آن منابع یا داده‌ها تنها زمانی بارگذاری می‌شوند که به آن‌ها نیاز باشد. این رویکرد باعث بهبود کارایی، کاهش زمان بارگذاری اولیه، و استفاده بهینه از حافظه می‌شود. در **Rust**، این الگو معمولاً با استفاده از کتابخانه‌هایی مانند lazy\_static یا ماکروهای جدیدتر مانند once\_cell پیاده‌سازی می‌شود.

### **پیاده‌سازی Lazy Loading با** lazy\_static

کتابخانه lazy\_static امکان تعریف متغیرهای ثابت را فراهم می‌کند که مقداردهی آن‌ها به صورت تنبل انجام می‌شود.

#### مثال:

rust

Copy code

#[macro\_use]

extern crate lazy\_static;

use std::collections::HashMap;

lazy\_static! {

static ref CONFIG: HashMap<String, String> = {

let mut m = HashMap::new();

println!("Initializing configuration...");

m.insert("username".to\_string(), "admin".to\_string());

m.insert("password".to\_string(), "1234".to\_string());

m

};

}

fn main() {

println!("Accessing configuration...");

// مقداردهی در این لحظه انجام می‌شود

println!("Username: {}", CONFIG.get("username").unwrap());

println!("Password: {}", CONFIG.get("password").unwrap());

}

#### **توضیح کد:**

1. متغیر CONFIG با lazy\_static! تعریف شده است.
2. مقداردهی اولیه زمانی انجام می‌شود که برای اولین بار به CONFIG دسترسی پیدا می‌کنیم.
3. پیام "Initializing configuration..." تنها یک بار نمایش داده می‌شود.

### **پیاده‌سازی Lazy Loading با** once\_cell

کتابخانه جدیدتر و بهینه‌تر once\_cell برای استفاده از بارگذاری تنبل پیشنهاد می‌شود.

#### مثال:

rust

Copy code

use once\_cell::sync::Lazy;

use std::collections::HashMap;

static CONFIG: Lazy<HashMap<String, String>> = Lazy::new(|| {

let mut m = HashMap::new();

println!("Initializing configuration...");

m.insert("username".to\_string(), "admin".to\_string());

m.insert("password".to\_string(), "1234".to\_string());

m

});

fn main() {

println!("Accessing configuration...");

println!("Username: {}", CONFIG.get("username").unwrap());

println!("Password: {}", CONFIG.get("password").unwrap());

}

### **جمع‌بندی**

Lazy Loading در Rust به شما اجازه می‌دهد داده‌ها یا منابع را به صورت تنبل مقداردهی کنید تا از کارایی بهتر و استفاده بهینه‌تر از حافظه بهره‌مند شوید. ابزارهایی مانند lazy\_static و once\_cell راهکارهای ساده و ایمن برای پیاده‌سازی این الگو ارائه می‌دهند.

**Lazy Loading** یا بارگذاری تنبل، روشی است که در آن مقداردهی به متغیرها یا بارگذاری منابع فقط در زمان نیاز انجام می‌شود، نه در زمان شروع برنامه. این روش باعث بهبود کارایی و کاهش مصرف منابع می‌شود. در **Rust** می‌توان با استفاده از کتابخانه‌هایی مانند lazy\_static یا once\_cell این مفهوم را پیاده‌سازی کرد.

### **مثال عملی از Lazy Loading با** once\_cell **در Rust**

rust

Copy code

use once\_cell::sync::Lazy;

use std::collections::HashMap;

// متغیر تنبل که فقط در اولین دسترسی مقداردهی می‌شود

static CONFIG: Lazy<HashMap<String, String>> = Lazy::new(|| {

println!("Initializing configuration...");

let mut m = HashMap::new();

m.insert("username".to\_string(), "admin".to\_string());

m.insert("password".to\_string(), "1234".to\_string());

m

});

fn main() {

println!("Before accessing configuration...");

// دسترسی به CONFIG باعث مقداردهی تنبل می‌شود

println!("Username: {}", CONFIG.get("username").unwrap());

println!("Password: {}", CONFIG.get("password").unwrap());

}

#### **توضیحات:**

1. از Lazy برای تعریف یک متغیر سراسری CONFIG استفاده کرده‌ایم.
2. مقداردهی به CONFIG فقط زمانی انجام می‌شود که اولین بار به آن دسترسی پیدا کنیم.
3. پیام "Initializing configuration..." فقط یک بار چاپ می‌شود، حتی اگر چندین بار به CONFIG دسترسی داشته باشیم.

### **خروجی برنامه**

yaml

Copy code

Before accessing configuration...

Initializing configuration...

Username: admin

Password: 1234

### **جمع‌بندی**

این مثال نشان می‌دهد که چگونه Rust با استفاده از الگوی Lazy Loading منابع را به صورت بهینه مقداردهی می‌کند و تنها در زمان نیاز از آن‌ها استفاده می‌کند، که منجر به بهبود کارایی و استفاده بهتر از حافظه می‌شود.

**سوال39:**

برنامه‌نویسی **آسنکرون (Asynchronous)** در **Rust** به توسعه برنامه‌هایی کمک می‌کند که می‌توانند بدون مسدود کردن نخ (thread) اصلی به طور همزمان چندین کار را انجام دهند. این امکان به لطف ویژگی‌های زبان و کتابخانه‌هایی مانند **async/await** و **Tokio** یا **async-std** فراهم شده است.

### **مفاهیم کلیدی**

* async: برای تعریف توابع آسنکرون استفاده می‌شود.
* await: برای منتظر ماندن بر روی نتیجه یک عملیات آسنکرون استفاده می‌شود.
* Future: نشان‌دهنده‌ی عملیاتی است که ممکن است بلافاصله کامل نشود.

### **مثال برنامه‌نویسی آسنکرون ساده**

استفاده از async-std برای یک درخواست آسنکرون:

rust

Copy code

use async\_std::task;

use std::time::Duration;

async fn greet(name: &str) {

task::sleep(Duration::from\_secs(2)).await;

println!("Hello, {}!", name);

}

fn main() {

let future = greet("Rust");

task::block\_on(future);

}

#### **توضیحات:**

1. greet یک تابع آسنکرون است که بعد از ۲ ثانیه چاپ می‌کند.
2. task::sleep یک عملیات آسنکرون برای تأخیر است.
3. task::block\_on اجرا را تا زمانی که future کامل شود مسدود می‌کند.

### **مثال پیچیده‌تر با** tokio **برای درخواست HTTP**

rust

Copy code

use tokio::time::{sleep, Duration};

async fn download\_file(url: &str) {

println!("Starting download from {}...", url);

sleep(Duration::from\_secs(3)).await;

println!("Download finished!");

}

#[tokio::main]

async fn main() {

let file1 = download\_file("http://example.com/file1");

let file2 = download\_file("http://example.com/file2");

tokio::join!(file1, file2); // اجرای همزمان هر دو دانلود

}

#### **توضیحات:**

1. tokio::sleep برای شبیه‌سازی زمان دانلود استفاده شده است.
2. tokio::join! برای اجرای همزمان چندین وظیفه استفاده می‌شود.
3. #[tokio::main] ماکرویی است که اجرای برنامه را تنظیم می‌کند.

### **مزایا**

* بهینه‌سازی مصرف منابع بدون نیاز به چند نخ (thread) برای هر عملیات.
* مناسب برای برنامه‌هایی مانند وب‌سرورها، پایگاه‌های داده، و کلاینت‌های شبکه.

### **جمع‌بندی**

برنامه‌نویسی آسنکرون در Rust به لطف ویژگی‌های async/await و کتابخانه‌هایی مانند **Tokio** بسیار قدرتمند و ایمن است. این رویکرد امکان استفاده مؤثر از منابع سیستم را فراهم می‌کند و برای برنامه‌های با کارایی بالا ایده‌آل است

**سوال40:**

در اینجا، ساختار داده **درخت جستجوی باینری** (Binary Search Tree) را در زبان **Rust** پیاده‌سازی می‌کنیم. این درخت برای ذخیره‌سازی و جستجوی داده‌ها با عملکرد کارآمد استفاده می‌شود. درخت جستجوی باینری این خاصیت را دارد که برای هر گره، مقدار تمام گره‌های سمت چپ کمتر و مقدار تمام گره‌های سمت راست بیشتر است.

**پیاده‌سازی درخت جستجوی باینری (BST) در Rust**

rust

Copy code

use std::cmp::Ordering;

use std::boxed::Box;

// تعریف یک گره (Node) درخت

#[derive(Debug)]

struct Node<T: Ord> {

value: T,

left: Option<Box<Node<T>>>,

right: Option<Box<Node<T>>>,

}

impl<T: Ord> Node<T> {

// ایجاد یک گره جدید

fn new(value: T) -> Self {

Node {

value,

left: None,

right: None,

}

}

// افزودن مقدار به درخت

fn insert(&mut self, new\_value: T) {

match new\_value.cmp(&self.value) {

Ordering::Less => {

if let Some(ref mut left) = self.left {

left.insert(new\_value);

} else {

self.left = Some(Box::new(Node::new(new\_value)));

}

}

Ordering::Greater => {

if let Some(ref mut right) = self.right {

right.insert(new\_value);

} else {

self.right = Some(Box::new(Node::new(new\_value)));

}

}

Ordering::Equal => {

// مقادیر برابر اضافه نمی‌شوند

println!("Value {} already exists in the tree.", new\_value);

}

}

}

// جستجوی مقدار در درخت

fn contains(&self, target: &T) -> bool {

match target.cmp(&self.value) {

Ordering::Less => {

if let Some(ref left) = self.left {

left.contains(target)

} else {

false

}

}

Ordering::Greater => {

if let Some(ref right) = self.right {

right.contains(target)

} else {

false

}

}

Ordering::Equal => true,

}

}

}

// تعریف یک درخت جستجوی باینری

#[derive(Debug)]

struct BinarySearchTree<T: Ord> {

root: Option<Box<Node<T>>>,

}

impl<T: Ord> BinarySearchTree<T> {

fn new() -> Self {

BinarySearchTree { root: None }

}

// افزودن مقدار به درخت

fn insert(&mut self, value: T) {

match self.root {

Some(ref mut root) => root.insert(value),

None => self.root = Some(Box::new(Node::new(value))),

}

}

// بررسی وجود مقدار

fn contains(&self, value: &T) -> bool {

match self.root {

Some(ref root) => root.contains(value),

None => false,

}

}

}

fn main() {

let mut bst = BinarySearchTree::new();

bst.insert(10);

bst.insert(5);

bst.insert(15);

bst.insert(5); // تکراری، اضافه نمی‌شود

println!("Tree contains 10: {}", bst.contains(&10));

println!("Tree contains 7: {}", bst.contains(&7));

println!("{:?}", bst); // نمایش ساختار درخت

}

**توضیحات**

* **Node**: هر گره شامل یک مقدار، یک فرزند چپ، و یک فرزند راست است.
* **BinarySearchTree**: درخت شامل یک ریشه است که به گره‌ها اشاره می‌کند.
* متدهای insert و contains برای افزودن و جستجو به کار می‌روند.
* از Box برای مدیریت مالکیت و تخصیص پویا استفاده شده است.

این پیاده‌سازی یک پایه برای BST است و می‌توان آن را با عملیات حذف، پیمایش، و بهینه‌سازی‌های دیگر توسعه داد

**سوال41:**

در اینجا یک پیاده‌سازی ساده و آموزشی از درخت AVL (درخت جستجوی باینری متوازن) را ارائه می‌دهم. این نوع درخت تفاوت ارتفاع زیرشاخه‌های هر گره را محدود می‌کند تا جستجو، درج، و حذف با پیچیدگی زمانی بهینه O(log⁡n)\mathcal{O}(\log n)O(logn) انجام شوند.

پیاده‌سازی درخت AVL در Rust

rust

Copy code

use std::cmp::{max, Ordering};

use std::boxed::Box;

#[derive(Debug)]

struct AVLNode<T: Ord> {

value: T,

height: i32,

left: Option<Box<AVLNode<T>>>,

right: Option<Box<AVLNode<T>>>,

}

impl<T: Ord> AVLNode<T> {

fn new(value: T) -> Self {

AVLNode {

value,

height: 1,

left: None,

right: None,

}

}

fn height(node: &Option<Box<AVLNode<T>>>) -> i32 {

match node {

Some(n) => n.height,

None => 0,

}

}

fn balance\_factor(&self) -> i32 {

AVLNode::height(&self.left) - AVLNode::height(&self.right)

}

fn update\_height(&mut self) {

self.height = 1 + max(AVLNode::height(&self.left), AVLNode::height(&self.right));

}

fn rotate\_right(mut self) -> Box<AVLNode<T>> {

let mut left\_node = self.left.take().unwrap();

self.left = left\_node.right.take();

self.update\_height();

left\_node.right = Some(Box::new(self));

left\_node.update\_height();

left\_node

}

fn rotate\_left(mut self) -> Box<AVLNode<T>> {

let mut right\_node = self.right.take().unwrap();

self.right = right\_node.left.take();

self.update\_height();

right\_node.left = Some(Box::new(self));

right\_node.update\_height();

right\_node

}

fn balance(mut self) -> Box<AVLNode<T>> {

self.update\_height();

let balance = self.balance\_factor();

if balance > 1 {

if self.left.as\_ref().unwrap().balance\_factor() < 0 {

self.left = Some(Box::new(self.left.take().unwrap().rotate\_left()));

}

return self.rotate\_right();

}

if balance < -1 {

if self.right.as\_ref().unwrap().balance\_factor() > 0 {

self.right = Some(Box::new(self.right.take().unwrap().rotate\_right()));

}

return self.rotate\_left();

}

Box::new(self)

}

fn insert(mut self, value: T) -> Box<AVLNode<T>> {

match value.cmp(&self.value) {

Ordering::Less => {

self.left = match self.left.take() {

Some(left) => Some(left.insert(value)),

None => Some(Box::new(AVLNode::new(value))),

};

}

Ordering::Greater => {

self.right = match self.right.take() {

Some(right) => Some(right.insert(value)),

None => Some(Box::new(AVLNode::new(value))),

};

}

Ordering::Equal => {}

}

self.balance()

}

}

#[derive(Debug)]

struct AVLTree<T: Ord> {

root: Option<Box<AVLNode<T>>>,

}

impl<T: Ord> AVLTree<T> {

fn new() -> Self {

AVLTree { root: None }

}

fn insert(&mut self, value: T) {

self.root = match self.root.take() {

Some(root) => Some(root.insert(value)),

None => Some(Box::new(AVLNode::new(value))),

};

}

}

fn main() {

let mut tree = AVLTree::new();

tree.insert(10);

tree.insert(20);

tree.insert(5);

tree.insert(15);

println!("{:#?}", tree); // نمایش درخت متوازن AVL

}

توضیحات:

* AVLNode: هر گره دارای مقدار، ارتفاع، و اشاره‌گرهایی به فرزندان چپ و راست است.
* متدهای rotate\_left و rotate\_right برای متوازن‌سازی درخت استفاده می‌شوند.
* balance بررسی و اصلاح تعادل گره را انجام می‌دهد.
* insert مقادیر جدید را اضافه و درخت را متوازن می‌کند.

این کد پیاده‌سازی اولیه است و می‌تواند با ویژگی‌های حذف و پیمایش بهبود یابد

**سوال42:**

در اینجا پیاده‌سازی یک Max Heap (درخت دودویی که در آن هر گره بزرگتر از فرزندان خود است) را در زبان Rust ارائه می‌دهیم. این ساختار داده برای الگوریتم‌هایی مانند مرتب‌سازی هیپ و صف‌های اولویت‌دار استفاده می‌شود.

پیاده‌سازی Max Heap در Rust

rust

Copy code

struct MaxHeap {

data: Vec<i32>, // ذخیره‌سازی عناصر در بردار

}

impl MaxHeap {

// ایجاد یک هیپ جدید

fn new() -> Self {

MaxHeap { data: Vec::new() }

}

// بازگرداندن اندازه هیپ

fn size(&self) -> usize {

self.data.len()

}

// درج یک عنصر در هیپ

fn insert(&mut self, value: i32) {

self.data.push(value); // افزودن به انتهای بردار

self.heapify\_up(self.size() - 1); // متوازن‌سازی

}

// حذف و بازگشت عنصر بیشینه

fn extract\_max(&mut self) -> Option<i32> {

if self.size() == 0 {

return None;

}

let max\_value = self.data.swap\_remove(0); // جایگزینی عنصر ریشه با آخرین عنصر

self.heapify\_down(0); // متوازن‌سازی

Some(max\_value)

}

// متوازن‌سازی به سمت بالا

fn heapify\_up(&mut self, mut index: usize) {

while index > 0 {

let parent = (index - 1) / 2;

if self.data[index] > self.data[parent] {

self.data.swap(index, parent);

index = parent;

} else {

break;

}

}

}

// متوازن‌سازی به سمت پایین

fn heapify\_down(&mut self, mut index: usize) {

let size = self.size();

loop {

let left = 2 \* index + 1;

let right = 2 \* index + 2;

let mut largest = index;

if left < size && self.data[left] > self.data[largest] {

largest = left;

}

if right < size && self.data[right] > self.data[largest] {

largest = right;

}

if largest == index {

break;

}

self.data.swap(index, largest);

index = largest;

}

}

}

fn main() {

let mut heap = MaxHeap::new();

heap.insert(10);

heap.insert(20);

heap.insert(5);

heap.insert(30);

heap.insert(15);

println!("Max element extracted: {:?}", heap.extract\_max());

println!("Max element extracted: {:?}", heap.extract\_max());

println!("Remaining heap: {:?}", heap.data);

}

توضیحات:

* MaxHeap: از یک بردار Vec برای ذخیره عناصر استفاده می‌شود.
* متدهای heapify\_up و heapify\_down برای حفظ ویژگی‌های هیپ به کار می‌روند.
* insert یک عنصر جدید را اضافه کرده و درخت را متوازن می‌کند.
* extract\_max عنصر بیشینه را حذف کرده و هیپ را دوباره متوازن می‌کند.

خروجی نمونه

yaml

Copy code

Max element extracted: Some(30)

Max element extracted: Some(20)

Remaining heap: [15, 10, 5]

این پیاده‌سازی را می‌توان با قابلیت‌های بیشتری مانند بررسی خالی بودن و بهینه‌سازی‌های دیگر گسترش داد

**سوال43:**برای پیاده‌سازی یک سرویس RESTful API در زبان Rust که درخواست‌های JSON را پردازش می‌کند، می‌توان از فریم‌ورک‌های محبوبی مانند Actix-web یا Rocket استفاده کرد. در اینجا با استفاده از Actix-web که از نظر عملکرد بسیار سریع و کاربرپسند است، یک مثال کامل ارائه می‌شود.

مثال: سرویس RESTful API با Actix-web

۱. تنظیمات پروژه با استفاده از Cargo.toml

toml

Copy code

[package]

name = "json\_api"

version = "0.1.0"

edition = "2021"

[dependencies]

actix-web = "4"

serde = { version = "1.0", features = ["derive"] }

serde\_json = "1.0"

۲. کد Rust برای API

rust

Copy code

use actix\_web::{web, App, HttpServer, Responder, HttpResponse, middleware};

use serde::{Deserialize, Serialize};

#[derive(Serialize, Deserialize)]

struct UserRequest {

name: String,

age: u8,

}

#[derive(Serialize)]

struct UserResponse {

message: String,

}

async fn process\_user(data: web::Json<UserRequest>) -> impl Responder {

let response = UserResponse {

message: format!("Hello, {}, you are {} years old!", data.name, data.age),

};

HttpResponse::Ok().json(response)

}

#[actix\_web::main]

async fn main() -> std::io::Result<()> {

HttpServer::new(|| {

App::new()

.wrap(middleware::Logger::default()) // فعال‌سازی لاگ‌ها

.route("/process\_user", web::post().to(process\_user)) // مسیر برای پردازش JSON

})

.bind("127.0.0.1:8080")? // اتصال به localhost:8080

.run()

.await

}

توضیحات

* این API یک نقطه پایان /process\_user دارد که درخواست‌های POST با بدنه JSON دریافت می‌کند.
* UserRequest و UserResponse از کتابخانه serde برای سریال‌سازی و دی‌سریال‌سازی استفاده می‌کنند.
* مسیر API توسط route ثبت شده است و تابع process\_user پاسخ را پردازش می‌کند.

ارسال درخواست نمونه با curl

bash

Copy code

curl -X POST -H "Content-Type: application/json" \

-d '{"name": "Alice", "age": 30}' \

http://127.0.0.1:8080/process\_user

خروجی پاسخ

json

Copy code

{

"message": "Hello, Alice, you are 30 years old!"

}

ویژگی‌ها و بهبودها

* می‌توانید مسیرهای دیگری برای افزودن یا حذف کاربران ایجاد کنید.
* Middleware برای احراز هویت یا مدیریت خطاها اضافه کنید.
* بهینه‌سازی برای تولید با استفاده از تنظیمات بیشتر Actix-web انجام دهید.

**سوال44:**

برای پیاده‌سازی یک سرویس ساده gRPC در زبان Rust می‌توان از کتابخانه **Tonic** استفاده کرد. این کتابخانه محبوب و کارآمد، سازگاری خوبی با اکوسیستم gRPC دارد. در ادامه، یک مثال ساده از یک سرور و کلاینت gRPC ارائه می‌شود.

**پیش‌نیازها**

۱. افزودن وابستگی‌ها به فایل Cargo.toml:

toml

Copy code

[dependencies]

tonic = "0.8"

prost = "0.11"

tokio = { version = "1", features = ["full"] }

[build-dependencies]

tonic-build = "0.8"

۲. نصب protoc و پلاگین protoc-gen-tonic برای کامپایل فایل‌های .proto.

**۱. تعریف فایل پروتکل service.proto**

proto

Copy code

syntax = "proto3";

package greeter;

service Greeter {

rpc SayHello (HelloRequest) returns (HelloReply);

}

message HelloRequest {

string name = 1;

}

message HelloReply {

string message = 1;

}

**۲. کامپایل فایل proto**

ایجاد اسکریپت build.rs برای کامپایل فایل proto:

rust

Copy code

fn main() -> Result<(), Box<dyn std::error::Error>> {

tonic\_build::compile\_protos("proto/service.proto")?;

Ok(())

}

**۳. کد Rust برای سرور gRPC**

rust

Copy code

use tonic::{transport::Server, Request, Response, Status};

use greeter::greeter\_server::{Greeter, GreeterServer};

use greeter::{HelloRequest, HelloReply};

pub mod greeter {

tonic::include\_proto!("greeter"); // تولید خودکار کدهای gRPC

}

#[derive(Default)]

pub struct MyGreeter {}

#[tonic::async\_trait]

impl Greeter for MyGreeter {

async fn say\_hello(&self, request: Request<HelloRequest>) -> Result<Response<HelloReply>, Status> {

let reply = HelloReply {

message: format!("Hello, {}!", request.into\_inner().name),

};

Ok(Response::new(reply))

}

}

#[tokio::main]

async fn main() -> Result<(), Box<dyn std::error::Error>> {

let addr = "[::1]:50051".parse()?;

let greeter = MyGreeter::default();

println!("GreeterServer listening on {}", addr);

Server::builder()

.add\_service(GreeterServer::new(greeter))

.serve(addr)

.await?;

Ok(())

}

**۴. کد Rust برای کلاینت gRPC**

rust

Copy code

use greeter::greeter\_client::GreeterClient;

use greeter::HelloRequest;

pub mod greeter {

tonic::include\_proto!("greeter");

}

#[tokio::main]

async fn main() -> Result<(), Box<dyn std::error::Error>> {

let mut client = GreeterClient::connect("http://[::1]:50051").await?;

let request = tonic::Request::new(HelloRequest {

name: "Alice".to\_string(),

});

let response = client.say\_hello(request).await?;

println!("Response: {:?}", response.into\_inner().message);

Ok(())

}

**توضیحات**

1. GreeterServer و GreeterClient با tonic\_build از فایل proto تولید می‌شوند.
2. سرور به درخواست SayHello پاسخ می‌دهد.
3. کلاینت پیام را به سرور می‌فرستد و پاسخ را نمایش می‌دهد.

این کد یک نمونه پایه برای گسترش با قابلیت‌های بیشتر است.

**سوال45:**

رای ایجاد یک سرویس ساده پردازش درخواست‌ها بر اساس WebAssembly (WASM) در زبان Rust، ابتدا نیاز به تنظیم محیط توسعه WebAssembly داریم. این کار معمولاً با کتابخانه‌هایی مثل wasm-bindgen و wasm-pack انجام می‌شود.مراحل پیاده‌سازی یک سرویس ساده WASM

۱. تنظیمات اولیه در Cargo.toml

toml

Copy code

[package]

name = "wasm\_service"

version = "0.1.0"

edition = "2021"

[lib]

crate-type = ["cdylib"]

[dependencies]

wasm-bindgen = "0.2"

۲. کد Rust برای تولید WebAssembly

rust

Copy code

use wasm\_bindgen::prelude::\*;

// این ماژول از JavaScript قابل فراخوانی است.

#[wasm\_bindgen]

pub fn process\_request(input: &str) -> String {

format!("Hello, {}! Your request is being processed.", input)

}

این تابع ورودی یک رشته را دریافت کرده و پاسخ ساده‌ای بازمی‌گرداند.

۳. ساخت WebAssembly با استفاده از wasm-pack

bash

Copy code

wasm-pack build --target web

۴. کد HTML و JavaScript برای استفاده از WebAssembly ایجاد یک فایل index.html:

html

Copy code

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="UTF-8">

<title>WASM Service Example</title>

</head>

<body>

<h1>WebAssembly Service</h1>

<input id="input" type="text" placeholder="Enter your name">

<button onclick="process()">Send Request</button>

<p id="output"></p>

<script type="module">

import init, { process\_request } from './pkg/wasm\_service.js';

async function process() {

await init(); // بارگذاری ماژول WebAssembly

const input = document.getElementById('input').value;

const output = process\_request(input);

document.getElementById('output').innerText = output;

}

</script>

</body>

</html>

توضیحات

1. wasm\_bindgen رابطی بین WebAssembly و JavaScript فراهم می‌کند.
2. wasm-pack فایل‌های WASM و کدهای جاوااسکریپت مربوطه را تولید می‌کند.
3. تابع process\_request در سمت کلاینت با کلیک دکمه فراخوانی می‌شود.

این سرویس ساده ورودی را پردازش و پاسخ مناسب تولید می‌کند.

**سوال 46:**

Socket Programming در زبان Rust برای ارتباط شبکه‌ای استفاده می‌شود و از کتابخانه استاندارد Rust برای سوکت‌ها و همچنین ابزارهای سطح بالاتر مانند Tokio و Mio برای برنامه‌نویسی ناهمگام بهره می‌گیرد. در اینجا یک مثال ساده از یک سرور و کلاینت TCP با استفاده از کتابخانه استاندارد Rust ارائه می‌شود.

. سرور TCP ساده در Rust

rust

Copy code

use std::io::{Read, Write};

use std::net::{TcpListener, TcpStream};

use std::thread;

fn handle\_client(mut stream: TcpStream) {

let mut buffer = [0; 512];

match stream.read(&mut buffer) {

Ok(\_) => {

let request = String::from\_utf8\_lossy(&buffer);

println!("Received: {}", request);

let response = format!("Hello, client! You sent: {}", request);

stream.write(response.as\_bytes()).unwrap();

}

Err(e) => eprintln!("Failed to read from socket: {}", e),

}

}

fn main() -> std::io::Result<()> {

let listener = TcpListener::bind("127.0.0.1:8080")?;

println!("Server is running on 127.0.0.1:8080");

for stream in listener.incoming() {

match stream {

Ok(stream) => {

thread::spawn(|| handle\_client(stream));

}

Err(e) => eprintln!("Connection failed: {}", e),

}

}

Ok(())

}

۲. کلاینت TCP ساده در Rust

rust

Copy code

use std::io::{self, Read, Write};

use std::net::TcpStream;

fn main() -> std::io::Result<()> {

let mut stream = TcpStream::connect("127.0.0.1:8080")?;

println!("Connected to the server");

let mut input = String::new();

println!("Enter a message to send:");

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

stream.write(input.as\_bytes())?;

let mut buffer = [0; 512];

stream.read(&mut buffer)?;

println!("Server response: {}", String::from\_utf8\_lossy(&buffer));

Ok(())

}

توضیحات

1. سرور با استفاده از TcpListener ایجاد شده و به پورت 8080 گوش می‌دهد.
2. هر اتصال جدید توسط handle\_client پردازش می‌شود. از thread::spawn برای مدیریت چندین اتصال همزمان استفاده شده است.
3. کلاینت به سرور متصل شده، یک پیام از کاربر می‌خواند و آن را به سرور ارسال می‌کند.
4. سرور پیام را دریافت کرده و پاسخ می‌دهد.

اجرای برنامه

1. ابتدا سرور را اجرا کنید:

bash

Copy code

cargo run --bin server

1. سپس کلاینت را اجرا کنید:

bash

Copy code

cargo run --bin client

این کد را می‌توان به راحتی برای برنامه‌نویسی ناهمگام با استفاده از Tokio بهبود داد

**سوال47:** برای نوشتن برنامه‌ای به زبان **Rust** که عملیات **CRUD** (ایجاد، خواندن، به‌روزرسانی و حذف) را روی پایگاه داده انجام دهد، می‌توان از کتابخانه **Diesel** یا **SQLx** استفاده کرد. در اینجا از **SQLx** که یکی از محبوب‌ترین ORMهای بدون نیاز به کد تولیدی (code generation) است، استفاده می‌کنیم.

### **. تنظیمات اولیه در** Cargo.toml

toml

Copy code

[dependencies]

sqlx = { version = "0.6", features = ["postgres", "runtime-tokio-native-tls"] }

tokio = { version = "1", features = ["full"] }

serde = { version = "1.0", features = ["derive"] }

**نکته**: فرض می‌کنیم از پایگاه داده PostgreSQL استفاده می‌کنید. می‌توانید تنظیمات مشابهی برای MySQL یا SQLite استفاده کنید.

### **۲. ساختار پایگاه داده**

فرض کنید جدولی به نام users با فیلدهای id, name, و email داریم.

sql

Copy code

CREATE TABLE users (

id SERIAL PRIMARY KEY,

name TEXT NOT NULL,

email TEXT NOT NULL

);

### **۳. برنامه Rust با عملیات CRUD**

#### **اتصال به پایگاه داده و تعریف عملیات CRUD**

rust

Copy code

use sqlx::{PgPool, FromRow};

use serde::{Deserialize, Serialize};

use std::env;

#[derive(Serialize, Deserialize, FromRow)]

struct User {

id: i32,

name: String,

email: String,

}

#[tokio::main]

async fn main() -> Result<(), sqlx::Error> {

let database\_url = env::var("DATABASE\_URL").expect("DATABASE\_URL must be set");

let pool = PgPool::connect(&database\_url).await?;

// ایجاد کاربر جدید

let new\_user = create\_user(&pool, "Alice", "alice@example.com").await?;

println!("Created user: {:?}", new\_user);

// خواندن اطلاعات کاربر

let user = get\_user\_by\_id(&pool, new\_user.id).await?;

println!("Fetched user: {:?}", user);

// به‌روزرسانی کاربر

let updated\_user = update\_user(&pool, user.id, "Alice Smith").await?;

println!("Updated user: {:?}", updated\_user);

// حذف کاربر

delete\_user(&pool, updated\_user.id).await?;

println!("Deleted user with ID: {}", updated\_user.id);

Ok(())

}

async fn create\_user(pool: &PgPool, name: &str, email: &str) -> Result<User, sqlx::Error> {

sqlx::query\_as!(

User,

"INSERT INTO users (name, email) VALUES ($1, $2) RETURNING id, name, email",

name,

email

)

.fetch\_one(pool)

.await

}

async fn get\_user\_by\_id(pool: &PgPool, user\_id: i32) -> Result<User, sqlx::Error> {

sqlx::query\_as!(User, "SELECT id, name, email FROM users WHERE id = $1", user\_id)

.fetch\_one(pool)

.await

}

async fn update\_user(pool: &PgPool, user\_id: i32, new\_name: &str) -> Result<User, sqlx::Error> {

sqlx::query\_as!(

User,

"UPDATE users SET name = $1 WHERE id = $2 RETURNING id, name, email",

new\_name,

user\_id

)

.fetch\_one(pool)

.await

}

async fn delete\_user(pool: &PgPool, user\_id: i32) -> Result<(), sqlx::Error> {

sqlx::query!("DELETE FROM users WHERE id = $1", user\_id)

.execute(pool)

.await?;

Ok(())

}

### **توضیحات**

1. از **SQLx** برای اجرای کوئری‌ها به صورت ناهمگام استفاده می‌شود.
2. توابع create\_user, get\_user\_by\_id, update\_user, و delete\_user عملیات CRUD را پیاده‌سازی می‌کنند.
3. اتصال به پایگاه داده با استفاده از DATABASE\_URL از متغیرهای محیطی انجام می‌شود.

### **اجرای برنامه**

1. متغیر محیطی DATABASE\_URL را تنظیم کنید:

bash

Copy code

export DATABASE\_URL=postgres://username:password@localhost/dbname

1. برنامه را اجرا کنید:

bash

Copy code

cargo run

این برنامه پایه‌ای است و می‌توان آن را برای قابلیت‌های بیشتر گسترش داد، مانند مدیریت خطا، لاگینگ و اعتبارسنجی داده‌ها.

**سوال48:** برای نوشتن یک برنامه در **Rust** که از ORM برای انجام عملیات **CRUD** روی پایگاه داده استفاده می‌کند، کتابخانه محبوب **Diesel** را معرفی می‌کنیم. این ORM از کدهای نوع ایمن (type-safe) استفاده می‌کند و پشتیبانی قوی برای پایگاه داده‌های مختلف مانند **PostgreSQL**، **SQLite** و **MySQL** ارائه می‌دهد.

### ****مراحل پیاده‌سازی CRUD با Diesel****

#### **۱. تنظیمات** Cargo.toml

toml

Copy code

[dependencies]

diesel = { version = "2.0", features = ["postgres", "r2d2", "chrono"] }

dotenvy = "0.15"

serde = { version = "1.0", features = ["derive"] }

**نکته:** فرض بر این است که از پایگاه داده PostgreSQL استفاده می‌کنید. برای MySQL یا SQLite باید ویژگی‌های مربوطه را تغییر دهید.

#### **۲. تنظیم Diesel**

اجرای دستور زیر برای تنظیم Diesel و ایجاد فایل‌های مورد نیاز:

bash

Copy code

diesel setup

#### **۳. تعریف مدل و اسکیما**

ایجاد فایل schema.rs و استفاده از ماکروی table!:

rust

Copy code

diesel::table! {

users (id) {

id -> Int4,

name -> Varchar,

email -> Varchar,

}

}

مدل User:

rust

Copy code

use diesel::prelude::\*;

use serde::{Deserialize, Serialize};

#[derive(Queryable, Serialize, Deserialize, Debug)]

pub struct User {

pub id: i32,

pub name: String,

pub email: String,

}

#[derive(Insertable)]

#[diesel(table\_name = users)]

pub struct NewUser<'a> {

pub name: &'a str,

pub email: &'a str,

}

#### **۴. پیاده‌سازی عملیات CRUD**

ایجاد فایل main.rs:

rust

Copy code

#[macro\_use]

extern crate diesel;

use diesel::prelude::\*;

use diesel::pg::PgConnection;

use dotenvy::dotenv;

use std::env;

mod schema;

mod models;

use models::{NewUser, User};

use schema::users;

fn establish\_connection() -> PgConnection {

dotenv().ok();

let database\_url = env::var("DATABASE\_URL").expect("DATABASE\_URL must be set");

PgConnection::establish(&database\_url).expect("Error connecting to the database")

}

fn create\_user(conn: &PgConnection, name: &str, email: &str) -> User {

use schema::users;

let new\_user = NewUser { name, email };

diesel::insert\_into(users::table)

.values(&new\_user)

.get\_result(conn)

.expect("Error saving new user")

}

fn main() {

let conn = establish\_connection();

// Create a new user

let user = create\_user(&conn, "Alice", "alice@example.com");

println!("Created user: {:?}", user);

}

### ****توضیحات****

1. Diesel از کدهای تولیدی برای اسکیما استفاده می‌کند.
2. توابع برای اتصال به پایگاه داده و ایجاد کاربر جدید استفاده می‌شوند.
3. عملیات خواندن، به‌روزرسانی، و حذف می‌توانند به صورت مشابه با استفاده از Diesel پیاده‌سازی شوند.

### ****اجرای برنامه****

1. تنظیم متغیر DATABASE\_URL برای اتصال به پایگاه داده:

bash

Copy code

export DATABASE\_URL=postgres://username:password@localhost/dbname

1. اجرای دستور:

bash

Copy code

cargo run

Diesel یک ORM قوی و نوع ایمن برای برنامه‌نویسی پایگاه داده در Rust است که با این رویکرد، به مدیریت کارآمد عملیات CRUD کمک می‌کند.

**سوال49:** در زبان **Rust**، چندین کتابخانه قدرتمند برای **Parsing** وجود دارد که به توسعه‌دهندگان کمک می‌کند تا متون پیچیده یا داده‌های ساختاریافته را تجزیه کنند. برخی از مهم‌ترین کتابخانه‌های Parsing عبارتند از:

### ****کتابخانه‌های Parsing محبوب در Rust****

1. **nom**: کتابخانه‌ای برای تجزیه‌گرهای کارآمد، نوع ایمن و تابعی.
2. **pest**: برای تجزیه‌گرهای قانون‌محور با قابلیت تعریف گرامر به راحتی.
3. **combine**: یک کتابخانه برای نوشتن پارسرهای ترکیبی.
4. **lalrpop**: یک مولد تجزیه‌گر LALR(1).
5. **peg**: یک کتابخانه برای ایجاد تجزیه‌گر با استفاده از گرامرهای PEG.

### ****مثال ساده با استفاده از کتابخانه**** nom

#### **مرحله 1: افزودن** nom **به** Cargo.toml

toml

Copy code

[dependencies]

nom = "7.1"

#### **مرحله 2: تعریف یک پارسر ساده برای اعداد صحیح**

این مثال یک رشته از اعداد صحیح را تجزیه می‌کند.

rust

Copy code

extern crate nom;

use nom::{

character::complete::{digit1, space0},

sequence::tuple,

IResult,

};

fn parse\_numbers(input: &str) -> IResult<&str, (u32, u32)> {

let parse\_tuple = tuple((digit1, space0, digit1));

let (remaining, (first, \_, second)) = parse\_tuple(input)?;

let first\_num = first.parse::<u32>().unwrap();

let second\_num = second.parse::<u32>().unwrap();

Ok((remaining, (first\_num, second\_num)))

}

fn main() {

let input = "42 78";

match parse\_numbers(input) {

Ok((\_, result)) => println!("Parsed numbers: {:?}", result),

Err(e) => eprintln!("Error parsing input: {:?}", e),

}

}

### ****توضیحات****

1. **parse\_numbers** تابعی است که از tuple و digit1 برای تجزیه رشته‌ای با دو عدد استفاده می‌کند.
2. tuple((digit1, space0, digit1)) الگوی دو عدد صحیح با یک فاصله بین آن‌ها را مشخص می‌کند.
3. پارسر از nom استفاده می‌کند که پارسرهای تابعی را به صورت ایمن و سریع فراهم می‌کند.
4. اگر ورودی مطابق گرامر باشد، اعداد صحیح استخراج و چاپ می‌شوند.

### ****خروجی برنامه****

bash

Copy code

Parsed numbers: (42, 78)

این مثال ساده قدرت و انعطاف‌پذیری کتابخانه‌های Parsing مانند nom را نشان می‌دهد که در کاربردهای مختلف مانند تجزیه فایل‌های پیکربندی یا زبان‌های دامنه‌محور (DSL) بسیار مفید است

**سوال50:**

**Regular Expression (Regex)** یک ابزار قدرتمند برای جستجو و پردازش الگوهای متنی است. با استفاده از عبارات منظم، می‌توان رشته‌ها را بر اساس الگوهای خاصی تطبیق داده و عملیات‌هایی مانند جستجو، جایگزینی و تقسیم‌بندی متن را انجام داد. در زبان **Rust**، از کتابخانه‌ی **regex** برای کار با عبارات منظم استفاده می‌شود.

### ****استفاده از Regex در Rust****

#### **افزودن وابستگی به** Cargo.toml

toml

Copy code

[dependencies]

regex = "1.7"

#### **مثال ۱: جستجوی یک الگو در یک رشته**

در این مثال، بررسی می‌کنیم که آیا یک رشته حاوی الگوی خاصی است یا خیر.

rust

Copy code

extern crate regex;

use regex::Regex;

fn main() {

let pattern = Regex::new(r"\b\d{4}\b").unwrap(); // الگو برای یافتن یک عدد ۴ رقمی

let text = "The year is 2025 and Rust is evolving.";

if pattern.is\_match(text) {

println!("Found a 4-digit number in the text.");

} else {

println!("No 4-digit number found.");

}

}

#### **توضیحات**

* Regex::new(r"\b\d{4}\b"): یک الگوی عبارات منظم برای جستجوی عددی که دقیقاً ۴ رقم دارد.
* is\_match(text): بررسی می‌کند که آیا متن شامل الگوی مشخص‌شده است یا خیر.

##### **خروجی**

arduino

Copy code

Found a 4-digit number in the text.

#### **مثال ۲: استخراج و چاپ تمام آدرس‌های ایمیل**

rust

Copy code

use regex::Regex;

fn main() {

let email\_pattern = Regex::new(r"[a-zA-Z0-9.\_%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}").unwrap();

let text = "Contact us at support@example.com or admin@domain.org for more info.";

for email in email\_pattern.find\_iter(text) {

println!("Found email: {}", email.as\_str());

}

}

##### **توضیحات**

* الگوی r"[a-zA-Z0-9.\_%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}" برای جستجوی آدرس‌های ایمیل.
* find\_iter(text) تمام تطابق‌ها را پیدا کرده و بازمی‌گرداند.

##### **خروجی**

graphql

Copy code

Found email: support@example.com

Found email: admin@domain.org

### ****توضیحات تکمیلی****

1. **عملیات تطبیق الگو**: تطبیق رشته‌ها با الگوهای پیچیده.
2. **گزاره‌های Regex**:
   * \d: عدد.
   * \b: مرز کلمه.
   * +: یک یا چند بار تکرار.
   * .: هر کاراکتری به‌جز خط جدید.

Rust با فراهم کردن یک کتابخانه سریع و ایمن برای Regex، استفاده از آن را برای کاربردهای مختلف ساده و کاربردی کرده است.

**سوال51:** کتابخانه **sysinfo** در زبان **Rust** اطلاعات مربوط به سیستم، از جمله استفاده از حافظه، پردازنده، فرآیندها، و دمای دستگاه را فراهم می‌کند. این کتابخانه برای نوشتن برنامه‌هایی که نیاز به نظارت بر منابع سیستم دارند، بسیار مفید است.

**نصب sysinfo**

ابتدا این وابستگی را در فایل Cargo.toml اضافه کنید:

toml

Copy code

[dependencies]

sysinfo = "0.29"

**مثال ساده: نمایش اطلاعات کلی سیستم**

rust

Copy code

use sysinfo::{System, SystemExt};

fn main() {

let mut system = System::new\_all();

// بارگذاری اطلاعات سیستم

system.refresh\_all();

// اطلاعات کلی سیستم

println!("Total memory: {} KB", system.total\_memory());

println!("Used memory: {} KB", system.used\_memory());

println!("Total swap: {} KB", system.total\_swap());

println!("Used swap: {} KB", system.used\_swap());

println!("Number of CPUs: {}", system.cpus().len());

println!("System name: {:?}", system.name().unwrap\_or\_else(|| "Unknown".to\_string()));

println!("Kernel version: {:?}", system.kernel\_version().unwrap\_or\_else(|| "Unknown".to\_string()));

println!("Uptime: {} seconds", system.uptime());

}

**توضیحات**

1. **System::new\_all()**: نمونه‌ای از سیستم ایجاد می‌کند و اطلاعات کامل را آماده می‌کند.
2. **refresh\_all()**: اطلاعات سیستم به‌روزرسانی می‌شود.
3. دستورات مختلف برای استخراج جزئیات حافظه، تعداد پردازنده‌ها، و زمان فعالیت (Uptime) استفاده می‌شوند.

**خروجی نمونه (به‌طور تقریبی)**

yaml

Copy code

Total memory: 16384000 KB

Used memory: 5200000 KB

Total swap: 8000000 KB

Used swap: 100000 KB

Number of CPUs: 4

System name: "Linux"

Kernel version: "5.15.0-53-generic"

Uptime: 3600 seconds

این مثال نشان می‌دهد که چگونه می‌توان از sysinfo برای نظارت و مدیریت منابع سیستم در برنامه‌های Rust استفاده کرد.

**سوال52:** کتابخانه‌های **native-windows-gui** و **windows** برای ایجاد برنامه‌های دسکتاپ و تعامل با سیستم‌عامل **ویندوز** در زبان **Rust** استفاده می‌شوند. این کتابخانه‌ها امکان ایجاد رابط‌های گرافیکی کاربر (GUI) و دسترسی به APIهای سیستمی ویندوز را فراهم می‌کنند.

### ****1. کتابخانه**** native-windows-gui

کتابخانه‌ای برای ساخت رابط‌های کاربری با استفاده از کنترل‌های استاندارد ویندوز.

#### **نصب وابستگی‌ها**

در فایل Cargo.toml:

toml

Copy code

[dependencies]

native-windows-gui = "1.0"

native-windows-derive = "1.0"

#### **مثال: ساخت یک پنجره ساده**

rust

Copy code

use native\_windows\_gui as nwg;

fn main() {

nwg::init().expect("Failed to initialize GUI");

let mut window = nwg::Window::default();

nwg::Window::builder()

.size((300, 150))

.position((300, 300))

.title("Hello, Windows!")

.build(&mut window)

.expect("Failed to create window");

nwg::dispatch\_thread\_events();

}

#### **توضیحات**

* nwg::init(): کتابخانه را مقداردهی اولیه می‌کند.
* nwg::Window::builder(): یک پنجره با اندازه و موقعیت مشخص ایجاد می‌کند.
* nwg::dispatch\_thread\_events(): مدیریت رویدادها را اجرا می‌کند.

### ****2. کتابخانه**** windows

این کتابخانه امکان دسترسی به **Windows API** را فراهم می‌کند.

#### **نصب وابستگی‌ها**

در فایل Cargo.toml:

toml

Copy code

[dependencies]

windows = { version = "0.48", features = ["Win32\_Foundation", "Win32\_UI\_WindowsAndMessaging"] }

#### **مثال: نمایش یک MessageBox**

rust

Copy code

use windows::Win32::Foundation::HWND;

use windows::Win32::UI::WindowsAndMessaging::{MessageBoxW, MB\_OK};

fn main() {

unsafe {

MessageBoxW(HWND(0), "Hello, Windows!".into(), "Greetings".into(), MB\_OK);

}

}

#### **توضیحات**

* MessageBoxW: تابع API ویندوز برای نمایش یک جعبه پیام.
* unsafe: برخی فراخوانی‌های سیستم به دلیل ایمنی حافظه نیاز به استفاده از بلوک ناامن دارند.

### ****تفاوت‌ها و کاربردها****

| **کتابخانه** | **هدف** | **کاربرد** |
| --- | --- | --- |
| native-windows-gui | رابط کاربری گرافیکی | ساخت برنامه‌های GUI ساده با کنترل‌های استاندارد |
| windows | دسترسی به Windows API | تعامل با توابع سیستمی ویندوز برای عملیات سطح پایین |

این دو کتابخانه به کاربران Rust کمک می‌کنند تا برنامه‌های دسکتاپ ویندوز با قابلیت‌های متفاوت ایجاد کنند

**سوال53:** برای انجام عملیات روی تصاویر در **Rust**، کتابخانه **image** یکی از محبوب‌ترین ابزارهاست. در اینجا یک مثال ساده برای بارگذاری یک تصویر، تغییر اندازه آن و ذخیره نتیجه ارائه می‌شود.

**نصب وابستگی‌ها**

اضافه کردن image به فایل Cargo.toml:

toml

Copy code

[dependencies]

image = "0.24"

**برنامه برای پردازش تصویر**

این برنامه یک تصویر را بارگذاری می‌کند، آن را به ابعاد جدید تغییر می‌دهد و نتیجه را ذخیره می‌کند.

rust

Copy code

extern crate image;

use image::GenericImageView;

fn main() {

let input\_path = "input.jpg";

let output\_path = "output\_resized.jpg";

// بارگذاری تصویر

let img = image::open(input\_path).expect("Failed to open image");

// نمایش ابعاد اصلی

println!("Original Dimensions: {:?}", img.dimensions());

// تغییر اندازه تصویر

let resized\_img = img.resize(200, 200, image::imageops::FilterType::Nearest);

// ذخیره تصویر تغییر یافته

resized\_img.save(output\_path).expect("Failed to save image");

println!("Image has been resized and saved as '{}'", output\_path);

}

**توضیحات**

1. image::open(input\_path): بارگذاری تصویر از فایل.
2. img.dimensions(): نمایش ابعاد اولیه تصویر.
3. resize(200, 200, FilterType::Nearest): تغییر اندازه تصویر به ابعاد ۲۰۰x۲۰۰ پیکسل.
4. save(output\_path): ذخیره تصویر به فایل جدید.

**ورودی و خروجی نمونه**

* **ورودی**: فایل تصویر input.jpg.
* **خروجی**: فایل تغییر اندازه‌یافته output\_resized.jpg.

**نتیجه**

این مثال ساده نشان می‌دهد که چگونه می‌توان عملیات پردازش اولیه تصویر مانند تغییر اندازه را با استفاده از زبان Rust و کتابخانه‌های مرتبط انجام داد. برای عملیات پیچیده‌تر می‌توانید از فیلترها یا ابزارهای دیگری در این کتابخانه استفاده کنید.

**سوال54:** برای پردازش فایل‌های صوتی در **Rust**، کتابخانه **hound** یکی از ابزارهای مناسب برای کار با فرمت **WAV** است. در اینجا یک مثال ساده برای بارگذاری یک فایل صوتی، افزایش حجم صدا و ذخیره آن به عنوان یک فایل جدید ارائه می‌شود.

**نصب وابستگی‌ها**

اضافه کردن hound به فایل Cargo.toml:

toml

Copy code

[dependencies]

hound = "3.4"

**برنامه برای پردازش صدا**

این برنامه فایل صوتی را بارگذاری می‌کند، سطح بلندی صدا را دو برابر می‌کند و آن را در یک فایل جدید ذخیره می‌کند.

rust

Copy code

extern crate hound;

fn main() {

let input\_path = "input.wav";

let output\_path = "output\_amplified.wav";

// بارگذاری فایل صوتی

let reader = hound::WavReader::open(input\_path).expect("Failed to open audio file");

let spec = reader.spec();

let samples: Vec<i16> = reader

.into\_samples::<i16>()

.map(|s| s.expect("Error reading sample"))

.map(|s| s.saturating\_mul(2)) // افزایش بلندی صدا با ضریب ۲

.collect();

// ذخیره فایل صوتی جدید

let mut writer = hound::WavWriter::create(output\_path, spec).expect("Failed to create output file");

for sample in samples {

writer.write\_sample(sample).expect("Failed to write sample");

}

println!("Audio processing complete. Output saved to '{}'", output\_path);

}

**توضیحات**

1. WavReader::open(input\_path): فایل صوتی با فرمت WAV را باز می‌کند.
2. into\_samples(): نمونه‌های صوتی را به بردار تبدیل می‌کند.
3. s.saturating\_mul(2): هر نمونه را دو برابر می‌کند تا حجم صدا افزایش یابد، از *saturating multiplication* برای جلوگیری از سرریز استفاده می‌شود.
4. WavWriter::create(output\_path, spec): فایل جدید را می‌سازد و نمونه‌ها را در آن می‌نویسد.

**ورودی و خروجی نمونه**

* **ورودی**: input.wav (یک فایل صوتی WAV).
* **خروجی**: output\_amplified.wav (فایل با صدای تقویت‌شده).

**کاربردها**

این روش برای اعمال افکت‌های ساده یا تغییر ویژگی‌های صوتی مفید است. برای پردازش پیچیده‌تر، می‌توانید از کتابخانه‌های دیگری مانند **rodio** یا **cpal** استفاده کنید

**سوال55:** پردازش ویدیو در **Rust** هنوز به اندازه زبان‌های دیگر مانند **Python** یا **C++** از پشتیبانی گسترده‌ای برخوردار نیست، اما با استفاده از کتابخانه‌هایی نظیر **ffmpeg** یا **gstreamer**، می‌توان به پردازش ویدیو پرداخت. در اینجا یک مثال ساده از استفاده از **ffmpeg** برای تغییر اندازه یک ویدیو ارائه می‌شود.

**مرحله 1: نصب وابستگی‌ها**

ابتدا مطمئن شوید که **FFmpeg** روی سیستم نصب شده است. سپس، کتابخانه **ffmpeg** را به فایل Cargo.toml اضافه کنید:

toml

Copy code

[dependencies]

ffmpeg-next = "4.4"

**برنامه: تغییر اندازه یک ویدیو**

rust

Copy code

extern crate ffmpeg\_next as ffmpeg;

use ffmpeg::{codec, format, frame, media, software, util};

fn main() -> Result<(), Box<dyn std::error::Error>> {

ffmpeg::init().unwrap();

let input\_path = "input.mp4";

let output\_path = "output\_resized.mp4";

let mut ictx = format::input(&input\_path)?;

let mut octx = format::output(&output\_path)?;

let input = ictx.streams().best(media::Type::Video).expect("No video stream found");

let input\_index = input.index();

let codec = input.codec();

let decoder = codec.decoder().video()?;

let mut scaler = software::scaling::context::Context::get(

decoder.format(),

decoder.width(),

decoder.height(),

decoder.format(),

640, // تغییر به عرض ۶۴۰

360, // تغییر به ارتفاع ۳۶۰

software::scaling::Flags::BILINEAR,

)?;

let mut frame = frame::Video::empty();

let mut packet = util::packet::Packet::empty();

let mut output = octx.add\_stream(codec.id())?;

output.set\_time\_base(input.time\_base());

let mut encoder = output.codec().encoder().video()?;

encoder.set\_height(360);

encoder.set\_width(640);

encoder.set\_format(decoder.format());

octx.write\_header()?;

while let Ok(\_) = ictx.read(&mut packet) {

if packet.stream() == input\_index {

decoder.decode(&packet, &mut frame)?;

let mut scaled = frame::Video::empty();

scaler.run(&frame, &mut scaled)?;

encoder.encode(&scaled, &mut packet)?;

octx.write\_frame(&mut packet)?;

}

}

octx.write\_trailer()?;

Ok(())

}

**توضیحات**

1. ffmpeg::init(): راه‌اندازی FFmpeg.
2. format::input(&input\_path): باز کردن فایل ورودی.
3. software::scaling::context::Context::get(): تغییر اندازه تصویر ویدیو.
4. encoder.encode() و octx.write\_frame(): کدگذاری و نوشتن فریم‌های جدید.

**خروجی**

* **ورودی**: input.mp4 (یک ویدیو).
* **خروجی**: output\_resized.mp4 با اندازه جدید.

این برنامه نیاز به دانش قبلی از کار با FFmpeg دارد و مناسب پردازش اولیه ویدیو در Rust است

**سوال56:** برای دانلود یک فایل از اینترنت در **Rust** می‌توان از کتابخانه **reqwest** استفاده کرد. این کتابخانه امکاناتی برای ارسال درخواست‌های HTTP فراهم می‌کند.

**نصب وابستگی‌ها**

اضافه کردن **reqwest** به فایل Cargo.toml:

toml

Copy code

[dependencies]

reqwest = { version = "0.11", features = ["blocking", "json"] }

**برنامه برای دانلود فایل**

این برنامه یک فایل از یک URL مشخص دانلود کرده و آن را در سیستم ذخیره می‌کند.

rust

Copy code

use std::fs::File;

use std::io::copy;

use std::path::Path;

use reqwest::blocking::get;

use std::error::Error;

fn main() -> Result<(), Box<dyn Error>> {

let url = "https://example.com/sample.txt"; // آدرس فایل برای دانلود

let output\_path = "downloaded\_sample.txt"; // نام فایل ذخیره‌شده

println!("Downloading from: {}", url);

let response = get(url)?;

let mut dest = File::create(&Path::new(output\_path))?;

copy(&mut response.bytes()?.as\_ref(), &mut dest)?;

println!("File has been downloaded to '{}'", output\_path);

Ok(())

}

**توضیحات**

1. get(url)?: ارسال درخواست HTTP GET به آدرس مشخص.
2. File::create(&Path::new(output\_path))?: ایجاد یک فایل برای ذخیره داده‌های دانلود شده.
3. copy(): کپی کردن محتوای دریافت‌شده به فایل مقصد.
4. مدیریت خطا با Result برای بازگشت مناسب در صورت شکست عملیات.

**ورودی و خروجی**

* **ورودی**: یک URL از فایل موجود در اینترنت.
* **خروجی**: ذخیره فایل دانلود شده در مسیر مشخص.

**توجه**

برای استفاده از **reqwest** ممکن است نیاز به نصب **OpenSSL** یا پیکربندی دیگر در سیستم داشته باشید. این روش برای فایل‌های بزرگ‌تر نیز کار می‌کند، اما می‌توانید از خواندن داده‌ها به صورت قطعه‌ای برای کاهش مصرف حافظه استفاده کنید

**سوال57:** برای خواندن و نوشتن فایل‌های CSV در **Rust**، می‌توان از کتابخانه **csv** استفاده کرد. این کتابخانه امکانات ساده‌ای برای مدیریت فایل‌های CSV فراهم می‌کند.

### ****نصب وابستگی‌ها****

اضافه کردن کتابخانه **csv** به فایل Cargo.toml:

toml

Copy code

[dependencies]

csv = "1.1"

serde = { version = "1.0", features = ["derive"] }

### ****برنامه برای خواندن و نوشتن CSV****

این مثال یک فایل CSV را می‌خواند و سپس یک فایل جدید با داده‌های اصلاح‌شده ایجاد می‌کند.

rust

Copy code

use csv::{ReaderBuilder, WriterBuilder};

use serde::{Deserialize, Serialize};

use std::error::Error;

#[derive(Debug, Deserialize, Serialize)]

struct Record {

name: String,

age: u8,

city: String,

}

fn main() -> Result<(), Box<dyn Error>> {

let input\_file = "input.csv";

let output\_file = "output.csv";

// خواندن داده‌ها از فایل CSV

let mut reader = ReaderBuilder::new().from\_path(input\_file)?;

let mut records = Vec::new();

for result in reader.deserialize() {

let record: Record = result?;

println!("{:?}", record);

records.push(record);

}

// نوشتن داده‌های اصلاح‌شده به فایل جدید

let mut writer = WriterBuilder::new().from\_path(output\_file)?;

for record in &records {

writer.serialize(record)?;

}

writer.flush()?;

println!("CSV data has been read and written successfully.");

Ok(())

}

### ****ورودی و خروجی نمونه****

#### input.csv

csv

Copy code

name,age,city

Alice,30,New York

Bob,25,Los Angeles

Charlie,22,Chicago

#### output.csv

داده‌های مشابه را در فایل جدید ذخیره می‌کند.

### ****توضیحات****

1. ReaderBuilder::new().from\_path(input\_file): فایل CSV را برای خواندن باز می‌کند.
2. deserialize(): داده‌ها را به ساختار Record تبدیل می‌کند.
3. WriterBuilder::new().from\_path(output\_file): فایل جدید برای نوشتن باز می‌شود.
4. serialize(): داده‌ها را به فرمت CSV می‌نویسد.

این روش برای کار با فایل‌های CSV به روشی ایمن و راحت استفاده می‌شود

**سوال58:** برای خواندن و نوشتن فایل‌های **Excel** در زبان **Rust** می‌توان از کتابخانه **calamine** استفاده کرد که از فرمت‌های **XLSX**، **XLS**، و **ODS** پشتیبانی می‌کند.

### ****نصب وابستگی‌ها****

اضافه کردن کتابخانه **calamine** به فایل Cargo.toml:

toml

Copy code

[dependencies]

calamine = "0.20"

### ****برنامه برای خواندن و نوشتن فایل Excel****

rust

Copy code

use calamine::{open\_workbook, Error, Reader, Xlsx, DataType, Writer, XlsxWriter};

fn main() -> Result<(), Error> {

// خواندن از فایل Excel

let mut workbook: Xlsx<\_> = open\_workbook("input.xlsx")?;

if let Some(Ok(range)) = workbook.worksheet\_range("Sheet1") {

for row in range.rows() {

for cell in row {

print!("{}\t", cell);

}

println!();

}

}

// نوشتن به فایل Excel جدید

let output\_path = "output.xlsx";

let mut writer = XlsxWriter::new()?;

let sheet\_name = "Sheet1";

writer.add\_sheet(sheet\_name);

writer.write\_cell(sheet\_name, 0, 0, DataType::String("Name".to\_string()))?;

writer.write\_cell(sheet\_name, 0, 1, DataType::String("Age".to\_string()))?;

writer.write\_cell(sheet\_name, 1, 0, DataType::String("Alice".to\_string()))?;

writer.write\_cell(sheet\_name, 1, 1, DataType::Int(30))?;

writer.close\_with\_path(output\_path)?;

println!("Excel read and write operations completed successfully.");

Ok(())

}

### ****توضیحات****

1. open\_workbook("input.xlsx"): فایل ورودی را باز می‌کند.
2. worksheet\_range("Sheet1"): داده‌ها را از شیت اول می‌خواند.
3. XlsxWriter::new(): ایجاد یک شیء برای نوشتن فایل جدید.
4. write\_cell(sheet\_name, row, col, DataType::String(...)): مقداردهی به سلول‌ها.

### ****ورودی و خروجی نمونه****

#### input.xlsx

| **Name** | **Age** |
| --- | --- |
| Bob | 25 |
| Eve | 28 |

#### output.xlsx

یک فایل جدید با داده‌های مثال در کد ذخیره می‌شود.

این رویکرد برای کارهای ساده و متوسط با فایل‌های Excel در Rust مناسب است

**سوال59:** برای پیاده‌سازی یک اپلیکیشن ساده مبتنی بر معماری **MVC (Model-View-Controller)** در **Rust**، می‌توان از چارچوب وب **actix-web** استفاده کرد که یکی از محبوب‌ترین فریمورک‌های Rust برای برنامه‌های تحت وب است. در اینجا یک مثال برای یک برنامه مدیریت کارهای روزانه (Todo) ارائه می‌شود.

### ****1. نصب وابستگی‌ها****

اضافه کردن وابستگی‌های مورد نیاز به Cargo.toml:

toml

Copy code

[dependencies]

actix-web = "4.0"

serde = { version = "1.0", features = ["derive"] }

serde\_json = "1.0"

### ****2. پیاده‌سازی مدل، ویو و کنترلر****

#### **مدل (Model): تعریف ساختار داده و منطق مدیریت**

rust

Copy code

use serde::{Deserialize, Serialize};

use std::sync::{Arc, Mutex};

#[derive(Serialize, Deserialize, Clone)]

struct Todo {

id: usize,

task: String,

done: bool,

}

struct AppState {

todos: Arc<Mutex<Vec<Todo>>>,

}

#### **کنترلر (Controller): مدیریت مسیرها و منطق پردازش درخواست‌ها**

rust

Copy code

use actix\_web::{web, App, HttpServer, Responder, HttpResponse};

async fn get\_todos(state: web::Data<AppState>) -> impl Responder {

let todos = state.todos.lock().unwrap();

HttpResponse::Ok().json(&\*todos)

}

async fn add\_todo(

state: web::Data<AppState>,

new\_todo: web::Json<Todo>,

) -> impl Responder {

let mut todos = state.todos.lock().unwrap();

todos.push(new\_todo.into\_inner());

HttpResponse::Created().finish()

}

async fn mark\_done(

state: web::Data<AppState>,

web::Path(id): web::Path<usize>,

) -> impl Responder {

let mut todos = state.todos.lock().unwrap();

if let Some(todo) = todos.iter\_mut().find(|todo| todo.id == id) {

todo.done = true;

}

HttpResponse::Ok().finish()

}

#### **ویو (View): تنظیم مسیرهای HTTP و اتصال آن‌ها به کنترلر**

rust

Copy code

use actix\_web::{web, App, HttpServer};

#[actix\_web::main]

async fn main() -> std::io::Result<()> {

let data = web::Data::new(AppState {

todos: Arc::new(Mutex::new(vec![])),

});

HttpServer::new(move || {

App::new()

.app\_data(data.clone())

.route("/todos", web::get().to(get\_todos))

.route("/todos", web::post().to(add\_todo))

.route("/todos/{id}/done", web::put().to(mark\_done))

})

.bind("127.0.0.1:8080")?

.run()

.await

}

### ****توضیحات****

1. مدل (Todo): ساختار داده‌ها و وضعیت اشتراکی.
2. کنترلرها: مدیریت درخواست‌های GET، POST، و PUT.
3. ویو: اتصال مسیرها به کنترلرها برای دسترسی به عملیات CRUD.

### ****اجرای برنامه****

برای اجرای برنامه، کافی است دستور زیر را وارد کنید:

sh

Copy code

cargo run

با ارسال درخواست‌های HTTP به آدرس http://127.0.0.1:8080/todos، می‌توانید لیست کارها را دریافت، اضافه، و به‌روز کنید

سوال60: پیاده‌سازی معماری پاک (Clean Architecture) در Rust نیازمند تقسیم برنامه به لایه‌های مستقل است تا هر لایه جدا از دیگری باشد. این معماری شامل لایه‌های زیر است:

1. Entities: قوانین تجاری و منطق برنامه.
2. Use Cases: عملیات و جریان‌های کاری برنامه.
3. Interface Adapters: تبدیل داده بین لایه‌ها.
4. Infrastructure: لایه‌ای که ارتباط با سیستم‌های خارجی را مدیریت می‌کند (پایگاه داده، سرویس‌های وب).

در اینجا مثالی از یک برنامه ساده مدیریت کارهای روزانه (Todo App) ارائه می‌شود.

ساختار پروژه

css

Copy code

src/

│

├── entities/

│ └── todo.rs

├── use\_cases/

│ └── manage\_todo.rs

├── interfaces/

│ └── todo\_repository.rs

├── infrastructure/

│ └── in\_memory\_repository.rs

├── main.rs

گام 1: تعریف موجودیت‌ها (Entities)

entities/todo.rs

rust

Copy code

#[derive(Clone, Debug)]

pub struct Todo {

pub id: usize,

pub task: String,

pub completed: bool,

}

گام 2: تعریف موارد استفاده (Use Cases)

use\_cases/manage\_todo.rs

rust

Copy code

use crate::entities::todo::Todo;

pub struct TodoManager {

pub todos: Vec<Todo>,

}

impl TodoManager {

pub fn new() -> Self {

Self { todos: vec![] }

}

pub fn add\_todo(&mut self, task: String) {

let id = self.todos.len() + 1;

let todo = Todo { id, task, completed: false };

self.todos.push(todo);

}

pub fn list\_todos(&self) -> &Vec<Todo> {

&self.todos

}

pub fn complete\_todo(&mut self, id: usize) {

if let Some(todo) = self.todos.iter\_mut().find(|t| t.id == id) {

todo.completed = true;

}

}

}

گام 3: رابط‌ها (Interfaces)

interfaces/todo\_repository.rs

rust

Copy code

use crate::entities::todo::Todo;

pub trait TodoRepository {

fn save(&self, todo: &Todo);

fn get\_all(&self) -> Vec<Todo>;

}

گام 4: زیرساخت‌ها (Infrastructure)

infrastructure/in\_memory\_repository.rs

rust

Copy code

use crate::entities::todo::Todo;

use crate::interfaces::todo\_repository::TodoRepository;

pub struct InMemoryTodoRepository {

pub todos: Vec<Todo>,

}

impl InMemoryTodoRepository {

pub fn new() -> Self {

Self { todos: vec![] }

}

}

impl TodoRepository for InMemoryTodoRepository {

fn save(&self, todo: &Todo) {

println!("Saving todo: {:?}", todo);

}

fn get\_all(&self) -> Vec<Todo> {

self.todos.clone()

}

}

گام 5: نقطه ورود اصلی

main.rs

rust

Copy code

mod entities;

mod use\_cases;

mod interfaces;

mod infrastructure;

use use\_cases::manage\_todo::TodoManager;

fn main() {

let mut manager = TodoManager::new();

manager.add\_todo("Learn Rust".to\_string());

manager.add\_todo("Build a project".to\_string());

println!("Todos:");

for todo in manager.list\_todos() {

println!("{:?}", todo);

}

manager.complete\_todo(1);

println!("After completing first task:");

for todo in manager.list\_todos() {

println!("{:?}", todo);

}

}

توضیحات

1. Entities: شامل ساختار Todo.
2. Use Cases: کلاس TodoManager عملیات افزودن، لیست‌کردن و تکمیل کارها را مدیریت می‌کند.
3. Interfaces: تعریف رابطی برای ذخیره داده‌ها.
4. Infrastructure: پیاده‌سازی یک مخزن داده در حافظه.

این معماری انعطاف‌پذیری و قابلیت نگهداری بالا را تضمین می‌کند

**سوال61:**

اصول **SOLID** پنج اصل برای طراحی بهتر و قابل نگهداری‌تر نرم‌افزار هستند. در اینجا هر اصل را توضیح داده و مثال‌هایی برای پیاده‌سازی آن در **Rust** ارائه می‌کنیم.

### ****1. Single Responsibility Principle (SRP)****

هر ماژول یا کلاس باید فقط یک مسئولیت داشته باشد.

#### مثال

rust

Copy code

struct FileReader;

impl FileReader {

pub fn read\_file(path: &str) -> Result<String, std::io::Error> {

std::fs::read\_to\_string(path)

}

}

struct DataProcessor;

impl DataProcessor {

pub fn process\_data(data: &str) {

println!("Processing data: {}", data);

}

}

اینجا FileReader مسئول خواندن فایل است و DataProcessor مسئول پردازش داده‌ها.

### ****2. Open/Closed Principle (OCP)****

ماژول‌ها باید برای توسعه باز و برای تغییر بسته باشند.

#### مثال

rust

Copy code

trait Shape {

fn area(&self) -> f64;

}

struct Circle {

radius: f64,

}

impl Shape for Circle {

fn area(&self) -> f64 {

3.1415 \* self.radius \* self.radius

}

}

struct Rectangle {

width: f64,

height: f64,

}

impl Shape for Rectangle {

fn area(&self) -> f64 {

self.width \* self.height

}

}

fn print\_area(shape: &dyn Shape) {

println!("Area: {}", shape.area());

}

افزودن شکل جدید نیاز به تغییر کدهای موجود ندارد، فقط نوع جدیدی از Shape تعریف می‌کنیم.

### ****3. Liskov Substitution Principle (LSP)****

هر کلاسی که از کلاس دیگری ارث‌بری می‌کند باید بتواند به جای آن استفاده شود.

#### مثال

rust

Copy code

trait Bird {

fn fly(&self);

}

struct Sparrow;

impl Bird for Sparrow {

fn fly(&self) {

println!("Sparrow is flying.");

}

}

fn let\_bird\_fly(bird: &dyn Bird) {

bird.fly();

}

let\_bird\_fly(&Sparrow);

### ****4. Interface Segregation Principle (ISP)****

کلاس‌ها نباید مجبور به پیاده‌سازی رابط‌هایی باشند که از آنها استفاده نمی‌کنند.

#### مثال

rust

Copy code

trait Printer {

fn print(&self);

}

trait Scanner {

fn scan(&self);

}

struct MultiFunctionPrinter;

impl Printer for MultiFunctionPrinter {

fn print(&self) {

println!("Printing...");

}

}

impl Scanner for MultiFunctionPrinter {

fn scan(&self) {

println!("Scanning...");

}

}

هر دستگاهی که فقط قابلیت چاپ یا اسکن دارد، نیازی به پیاده‌سازی همه متدها ندارد.

### ****5. Dependency Inversion Principle (DIP)****

کلاس‌ها باید به انتزاعات وابسته باشند نه به پیاده‌سازی‌ها.

#### مثال

rust

Copy code

trait Database {

fn save(&self, data: &str);

}

struct MySQL;

impl Database for MySQL {

fn save(&self, data: &str) {

println!("Saving '{}' to MySQL database", data);

}

}

struct DataSaver<'a> {

db: &'a dyn Database,

}

impl<'a> DataSaver<'a> {

fn new(db: &'a dyn Database) -> Self {

Self { db }

}

fn save(&self, data: &str) {

self.db.save(data);

}

}

fn main() {

let mysql = MySQL;

let saver = DataSaver::new(&mysql);

saver.save("Example data");

}

اینجا DataSaver به انتزاع Database وابسته است، نه به کلاس خاصی مثل MySQL.

با استفاده از اصول **SOLID**، کدهای شما خواناتر، قابل تغییرتر و ماژولارتر خواهند بود